InferPy Documentation

Release 1.0

Rafael Cabañas

Aug 03, 2018

Quick Start

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

Use InferPy is 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.

CHAPTER 1

InferPy is to Edward what Keras is to Tensorflow

InferPy's aim is to be to Edward what Keras is to Tensorflow. Edward is a general purpose probabilistic programing language, like Tensorflow is a general computational engine. But this generality comes a at price. Edward's API is verbose and is based on distributions over Tensor objects, which are n-dimensional arrays with complex semantics operations. Probability distributions over Tensors are powerful abstractions but it is not easy to operate with them. InferPy's API is no so general like Edward's API but still covers a wide range of powerful and widely used probabilistic models, which can contain complex probability constructs.

1.1 Getting Started:

1.1.1 Installation

Install InferPy from PyPI:

```
$ pip install inferpy
```

1.1.2 30 seconds to InferPy

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

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

We start defining the **prior** of the global parameters,

```
import inferpy as inf
from inferpy.models import Normal

# K defines the number of components.
K=10
```

(continues on next page)

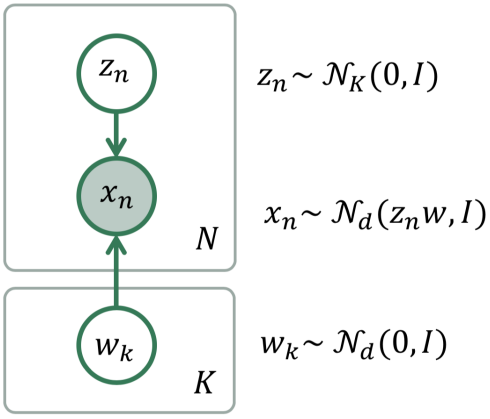

Fig. 1: Bayesian PCA

(continued from previous page)

```
# d defines the number of dimensions
d=20

#Prior for the principal components
with inf.replicate(size = K):
    w = Normal(loc = 0, scale = 1, dim = d)
```

InferPy supports the definition of **plateau notation** by using the construct `with inf.replicate(size = K)`, which replicates K times the random variables enclosed within this anotator. Every replicated variable is assumed to be **independent**.

This `with inf.replicate(size = N)` construct is also useful when defining the model for the data:

```
# Number of observations
N = 1000

# define the generative model
with inf.replicate(size=N):
    z = Normal(0, 1, dim=K)
    x = Normal(inf.matmul(z,w), 1.0, observed=True, dim=d)
```

As commented above, the variables are surrounded by a `with` statement to inidicate that the defined random variables will be reapeatedly used in each data sample. In this case, every replicated variable is conditionally idependent given the variables mu and sigma defined outside the `with` statement.

Once the random variables of the model are defined, the probablitic model itself can be created and compiled. The probabilistic model defines a joint probability distribuiton over all these random variables.

```
from inferpy import ProbModel

# Define the model
pca = ProbModel(varlist = [w,z,x])

# Compile the model
pca.compile(infMethod = 'KLqp')
```

During the model compilation we specify different inference methods that will be used to learn the model.

```
from inferpy import ProbModel

# Define the model
pca = ProbModel(varlist = [w,z,x])

# Compile the model
pca.compile(infMethod = 'Laplace')
```

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

Every random variable object is equipped with methods such as `log_prob()` and `sample()`. Similarly, a probabilistic model is also equipped with the same methods. Then, we can sample data from the model anbd compute the log-likelihood of a data set:

```
# Sample data from the model
data = pca.sample(size = 100)

# Compute the log-likelihood of a data set
log_like = pca.log_prob(data)
```

Of course, you can fit your model with a given data set:

```
# compile and fit the model with training data
pca.compile()
pca.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = pca.posterior(z)
```

1.2 Guiding Principles

- InferPy's probability distribuions are mainly inherited from TensorFlow Distribuitons package. .. InferPy's API is fully compatible with tf.distributions' API. The 'shape' argument was added as a simplifing option when defining multidimensional distributions.
- 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 seamsly 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 and Apache Spark computing engines.

1.3 Guide to Building Probabilistic Models

1.3.1 Getting Started with Probabilistic Models

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

- A **prior model** defining a joint distribution $p(\theta)$ over the global parameters of the model, θ.
- A **data or observation model** defining a joint conditional distribution $p(x, h|\theta)$ over the observed quantities x and the the local hidden variables h governing the observation x. This data model should be specified in a single-sample basis. There are many models of interest without local hidden variables, in that case we simply specify the conditional $p(x|\theta)$. More flexible ways of defining the data model can be found in ?.

A Bayesian PCA model has the following graphical structure,

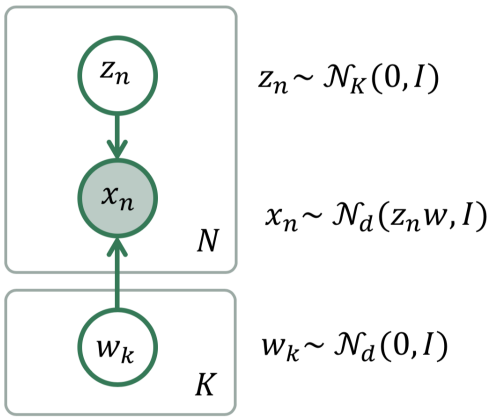

Fig. 2: Bayesian PCA

The **prior model** is the variable μ. 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 edward as ed
import inferpy as inf
from inferpy.models import Normal

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        z = Normal(0, 1, dim=K)
        x = Normal(inf.matmul(z,w), 1.0, observed=True, dim=d)

m.compile()
```

The `with inf.replicate(size = N)` 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.

1.3.2 Random Variables

Following Edward's approach, a random variable x is an object parametrized by a tensor θ (i.e. a TensorFlow's tensor or numpy's ndarray). The number of random variables in one object is determined by the dimensions of its parameters (like in Edward) or by the 'dim' argument (inspired by PyMC3 and Keras):

```
import inferpy as inf
import tensorflow as tf
import numpy as np


# different ways of declaring 1 batch of 5 Normal distributions

x = inf.models.Normal(loc = 0, scale=1, dim=5)          # x.shape = [5]

x = inf.models.Normal(loc = [0, 0, 0, 0, 0], scale=1)   # x.shape = [5]

x = inf.models.Normal(loc = np.zeros(5), scale=1)       # x.shape = [5]

x = inf.models.Normal(loc = 0, scale=tf.ones(5))        # x.shape = [5]
```

The `with inf.replicate(size = N)` sintaxis can also be used to define multi-dimensional objects:

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

Following Edward's approach, the multivariate dimension is the innermost (right-most) dimension of the parameters.

The argument `observed = true` in the constructor of a random variable is used to indicate whether a variable is observable or not.

1.3.3 Probabilistic Models

A **probabilistic model** defines a joint distribution over observable and non-observable variables, $p(\theta, \mu, \sigma, z_n, x_n)$ for the running example. The variables in the model are the ones defined using the `with inf.ProbModel() as pca:` construct. Alternatively, we can also use a builder,

```
m = inf.ProbModel(varlist=[w,z,x])
m.compile()
```

The model must be **compiled** before it can be used.

Like any random variable object, a probabilistic model is equipped with methods such as `sample()`, `log_prob()` and `sum_log_prob()`. Then, we can sample data from the model and compute the log-likelihood of a data set:

```
data = m.sample(1000)
log_like = m.log_prob(data)
sum_log_like = m.sum_log_prob(data)
```

Folowing Edward's approach, a random variable x is associated to a tensor x^* in the computational graph handled by TensorFlow, where the computations takes place. This tensor x^* contains the samples of the random variable x, i.e. $x^* \sim p(x|\theta)$. In this way, random variables can be involved in expressive deterministic operations.

Dependecies between variables are modelled by setting a given variable as a parameter of another variable. For example:

```
with inf.ProbModel() as m:
    theta = inf.models.Beta(0.5,0.5)
    z =  inf.models.Categorical(probs=[theta, 1-theta], name="z")


m.sample()
```

Moreover, we might consider using the function `inferpy.case` as the parameter of other random variables:

```
# Categorical variable depending on another categorical variable

with inf.ProbModel() as m2:
    y =  inf.models.Categorical(probs=[0.4,0.6], name="y")
    x = inf.models.Categorical(probs=inf.case({y.equal(0): [0.0, 1.0],
                                               y.equal(1): [1.0, 0.0] }), name="x")
m2.sample()


# Categorical variable depending on a Normal distributed variable

with inf.ProbModel() as m3:
    a = inf.models.Normal(0,1, name="a")
    b = inf.models.Categorical(probs=inf.case({a>0: [0.0, 1.0],
                                               a<=0: [1.0, 0.0]}), name="b")
m3.sample()


# Normal distributed variable depending on a Categorical variable

with inf.ProbModel() as m4:
    d =  inf.models.Categorical(probs=[0.4,0.6], name="d")
    c = inf.models.Normal(loc=inf.case({d.equal(0): 0.,
                                        d.equal(1): 100.}), scale=1., name="c")
m4.sample()
```

1.3.4 Supported Probability Distributions

Supported probability distributions are located in the package `inferpy.models`. All of them have `inferpy.models.RandomVariable` as superclass. Those currently implemented are:

```
>>> inf.models.ALLOWED_VARS
['Bernoulli', 'Beta', 'Categorical', 'Deterministic', 'Dirichlet', 'Exponential',
↪'Gamma', 'InverseGamma', 'Laplace', 'Multinomial', 'Normal', 'Poisson', 'Uniform']
```

1.4 Guide to Approximate Inference

1.4.1 Getting Started with Approximate 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 variibles, 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. For the running example, 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.

A probabilistic model in InferPy should be compiled before we can access these posteriors,

```
m.compile(infMethod="KLqp")
m.fit(x_train)
m.posterior(z)
```

The compilation process allows to choose the inference algorithm through the `infMethod` argument. In the above example we use `'Klqp'`.

Following InferPy guiding principles, users can further configure the inference algorithm. First, they can define a model 'Q' for approximating the posterior distribution,

```
qw = inf.Qmodel.Normal(w)
qz = inf.Qmodel.Normal(z)

qmodel = inf.Qmodel([qw, qz])

m.compile(infMethod="KLqp", Q=qmodel)
m.fit(x_train)
m.posterior(z)
```

In the 'Q' model we should include a q distribution for every non observed variable in the 'P' model. Otherwise, an error will be raised during model compilation.

By default, the posterior **q** belongs to the same distribution family than **p** , but in the above example we show how we can change that (e.g. we set the posterior over **mu** to obtain a point mass estimate instead of the Gaussian approximation used by default). We can also configure how these **q's** are initialized using any of the Keras's initializers.

1.4.2 Compositional Inference

Note: not implemented yet

InferPy directly builds on top of Edward's compositionality idea to design complex infererence algorithms.

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.PointMass(bind = mu, initializer='zeroes')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qlocal = QModel(vars = [q_w_n])
qglobal = QModel(vars = [mu])

infkl_qp = inf.inference.KLqp(Q = qlocal, optimizer = 'sgd', innerIter = 10)
infMAP = inf.inference.MAP(Q = qglobal, optimizer = 'sgd')
```

(continues on next page)

(continued from previous page)

```
sgd = keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True)
infkl_qp = inf.inference.KLqp(Q = qmodel, optimizer = sgd, loss="ELBO")
probmodel.compile(infMethod = [infkl_qp,infMAP])

pca.fit(x_train)
posterior_mu = pca.posterior(mu)
```

With the above sintaxis, we perform a variational EM algorithm, where the E step is repeated 10 times for every MAP step.

More flexibility is also available by defining how each mini-batch is processed by the inference algorithm. The following piece of code is equivalent to the above one,

```
pca = ProbModel(vars = [mu,w_n,x_n])

q_mu = inf.inference.Q.PointMass(bind = mu, initializer='zeroes')
q_w_n = inf.inference.Q.Normal(bind = w_n, initializer='random_unifrom')

qlocal = QModel(vars = [q_w_n])
qglobal = QModel(vars = [mu])

infkl_qp = inf.inference.KLqp(Q = qlocal, optimizer = 'sgd', innerIter = 10)
infMAP = inf.inference.MAP(Q = qglobal, optimizer = 'sgd')

emAlg = lambda (infMethod, dataBatch):
   for _ in range(10)
       infMethod[0].update(data = dataBatch)

   infMethod[1].update(data = dataBatch)
   return

pca.compile(infMethod = [infkl_qp,infMAP], ingAlg = emAlg)

pca.fit(x_train, EPOCHS = 10)
posterior_mu = pca.posterior(mu)
```

Have a look again at Inference Zoo to explore other complex compositional options.

1.4.3 Supported Inference Methods

1.5 Guide to Model Validation

Note: not implemented yet

Model validation try to assess how faifhfully the inferered probabilistic model represents and explain the observed data.

The main tool for model validation consists on analyzing the posterior predictive distribution,

$p(y_{test}, x_{test}|y_{train}, x_{train}) = \int p(y_{test}, x_{test}|z, \theta)p(z, \theta|y_{train}, x_{train})dzd\theta$

This posterior predictive distribution can be used to measure how well the model fits an independent dataset using the test marginal log-likelihood, $\ln p(y_{test}, x_{test}|y_{train}, x_{train})$,

```
log_like = probmodel.evaluate(test_data, metrics = ['log_likelihood'])
```

In other cases, we may need to evalute the predictive capacity of the model with respect to some target variable y,

$p(y_{test}|x_{test}, y_{train}, x_{train}) = \int p(y_{test}|x_{test}, z, \theta)p(z, \theta|y_{train}, x_{train})dzd\theta$

So the metrics can be computed with respect to this target variable by using the 'targetvar' argument,

```
log_like, accuracy, mse = probmodel.evaluate(test_data, targetvar = y, metrics = [
↪'log_likelihood', 'accuracy', 'mse'])
```

So, the log-likelihood metric as well as the accuracy and the mean square error metric are computed by using the predictive posterior $p(y_{test}|x_{test}, y_{train}, x_{train})$.

Custom evaluation metrics can also be defined,

```
def mean_absolute_error(posterior, observations, weights=None):
    predictions = tf.map_fn(lambda x : x.getMean(), posterior)
    return tf.metrics.mean_absolute_error(observations, predictions, weights)

mse, mae = probmodel.evaluate(test_data, targetvar = y, metrics = ['mse', mean_
↪absolute_error])
```

1.6 Guide to Data Handling

InferPy leverages existing Pandas functionality for reading data. As a consequence, InferPy can learn from datasets in any file format handled by Pandas. This is possible because the method `inferpy.ProbModel.fit(data)` accepts as input argument a Pandas DataFrame.

In the following code fragment, an example of learning a model from a CVS file is shown:

```
import inferpy as inf
import pandas as pd


data = pd.read_csv("inferpy/datasets/test.csv")
N = len(data)

with inf.ProbModel() as m:

    thetaX = inf.models.Normal(loc=0., scale=1.)
    thetaY = inf.models.Normal(loc=0., scale=1.)


    with inf.replicate(size=N):
        x = inf.models.Normal(loc=thetaX, scale=1., observed=True, name="x")
        y = inf.models.Normal(loc=thetaY, scale=1., observed=True, name="y")


m.compile()
m.fit(data)

m.posterior([thetaX, thetaY])
```

1.7 Probabilistic Model Zoo

1.7.1 Bayesian Linear Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal

d, N =  10, 200

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1)
    with inf.replicate(size=d):
        w = Normal(0, 1)

    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        y = Normal(w0 + inf.matmul(x,w), 1.0, observed=True)

# toy data generation
x_train = Normal(loc=0, scale=1, dim=d).sample(N)
y_train = Normal(loc=5, scale=1, dim=1).sample(N)
data = {x.name: x_train, y.name: y_train}


# compile and fit the model with training data
m.compile()
m.fit(data)

print(m.posterior([w, w0]))
```

1.7.2 Bayesian Logistic Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, Bernoulli, Categorical
import numpy as np

d, N =  10, 500

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1)
    with inf.replicate(size=d):
        w = Normal(0, 1)
```

(continues on next page)

(continued from previous page)

```
    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        p = w0 + inf.matmul(x, w)
        y = Bernoulli(logits = p, observed=True)


# toy data generation
x_train = Normal(loc=0, scale=1, dim=d).sample(N)
y_train = Bernoulli(probs=[0.4]).sample(N)
data = {x.name: x_train, y.name: np.reshape(y_train, (N,1))}


# compile and fit the model with training data
m.compile()
m.fit(data)

print(m.posterior([w, w0]))
```

1.7.3 Bayesian Multinomial Logistic Regression

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, Bernoulli, Categorical
import numpy as np

d, N =  10, 500

#number of classes
K = 3

# model definition
with inf.ProbModel() as m:

    #define the weights
    w0 = Normal(0,1, dim=K)

    with inf.replicate(size=d):
        w = Normal(0, 1, dim=K)

    # define the generative model
    with inf.replicate(size=N):
        x = Normal(0, 1, observed=True, dim=d)
        p = w0 + inf.matmul(x, w)
        y = Bernoulli(logits = p, observed=True)


# toy data generation
x_train = Normal(loc=0, scale=1, dim=d).sample(N)
y_train = Bernoulli(probs=np.random.rand(K)).sample(N)
```

(continues on next page)

(continued from previous page)

```
data = {x.name: x_train, y.name: np.reshape(y_train, (N,K))}


# compile and fit the model with training data
m.compile()
m.fit(data)

print(m.posterior([w, w0]))
```

1.7.4 Linear Factor Model (PCA)

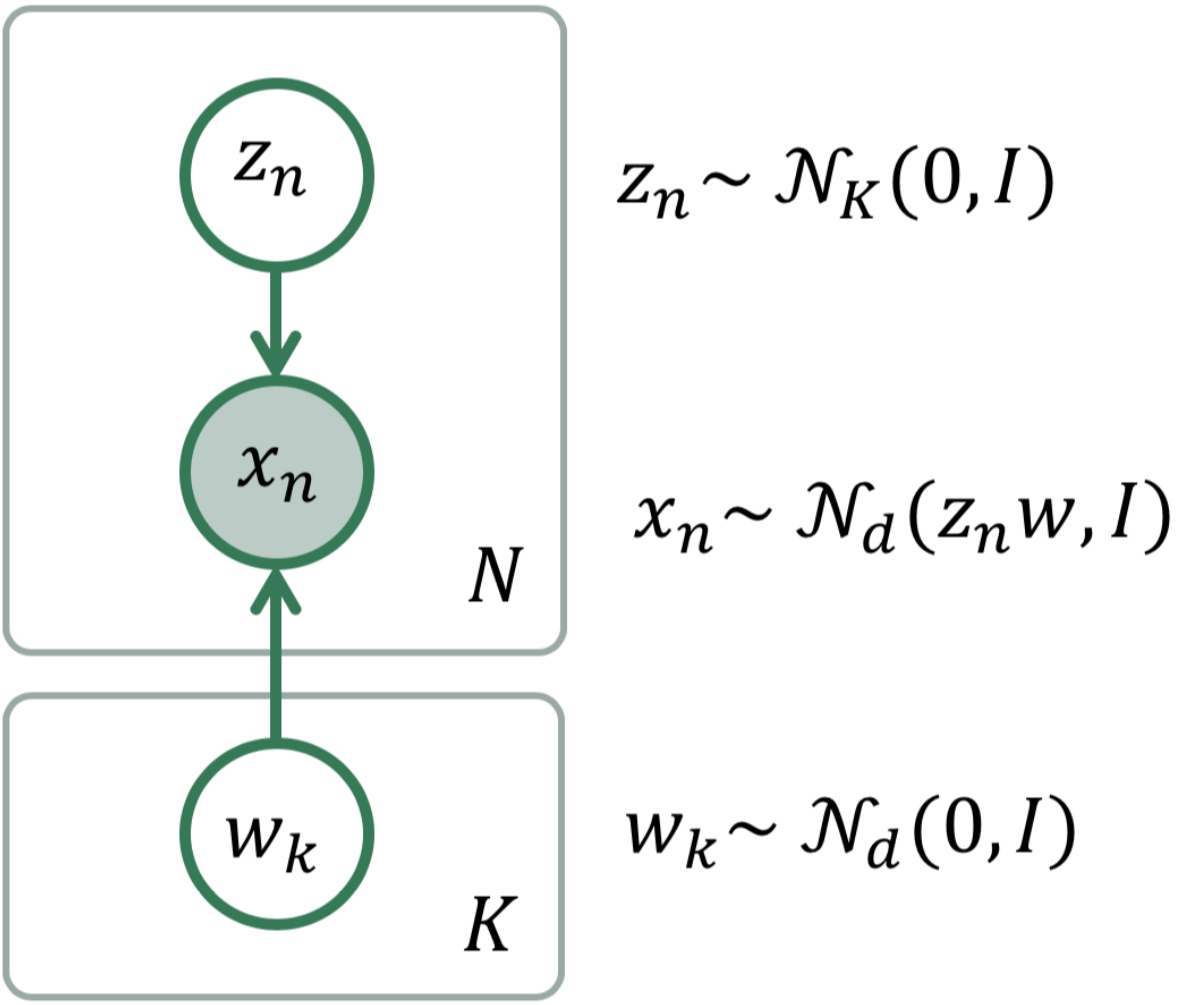

Fig. 3: Linear Factor Model

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = Normal(0, 1, dim=d)

    # define the generative model
    with inf.replicate(size=N):
        z = Normal(0, 1, dim=K)
        x = Normal(inf.matmul(z,w),
                   1.0, observed=True, dim=d)

# toy data generation
x_train = Normal(loc=0, scale=1., dim=d).sample(N)
data = {x.name: x_train}


# compile and fit the model with training data
m.compile()
m.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = m.posterior(z)
```

1.7.5 PCA with ARD Prior (PCA)

```
import edward as ed
import inferpy as inf
from inferpy.models import Normal, InverseGamma

K, d, N = 5, 10, 200

# model definition
with inf.ProbModel() as m:
    #define the weights
    with inf.replicate(size=K):
        w = Normal(0, 1, dim=d)

    sigma = InverseGamma(1.0,1.0)

    # define the generative model
    with inf.replicate(size=N):
        z = Normal(0, 1, dim=K)
        x = Normal(inf.matmul(z,w),
                   sigma, observed=True, dim=d)

# toy data generation
```

(continues on next page)

(continued from previous page)

```
x_train = Normal(loc=0, scale=1., dim=d).sample(N)
data = {x.name: x_train}


# compile and fit the model with training data
m.compile()
m.fit(data)

#extract the hidden representation from a set of observations
hidden_encoding = m.posterior(z)
```

1.7.6 Matrix Factorization

```
import inferpy as inf
from inferpy.models import Normal

N=200
M=50
K=5

# Shape [M,K]
with inf.replicate(size=K):
    gamma = Normal(0,1, dim=M)

# Shape [N,K]
with inf.replicate(size=N):
    w = Normal(0,1, dim=K)

# x_mn has shape [N,K] x [K,M] = [N,M]

with inf.replicate(size=N):
    x = Normal(inf.matmul(w,gamma), 1, observed = True)

m = inf.ProbModel([w,gamma,x])

data = m.sample(size=N)

log_prob = m.log_prob(data)

m.compile(infMethod = 'KLqp')

m.fit(data)

print(m.posterior([w,gamma]))
```

1.8 inferpy package

1.8.1 Subpackages

inferpy.inferences package

Module contents

inferpy.models package

Submodules

inferpy.models.normal module

inferpy.models.random_variable module

Module contents

inferpy.util package

Submodules

inferpy.util.error module

inferpy.util.ops module

inferpy.util.runtime module

inferpy.util.wrappers module

Module contents

1.8.2 Submodules

1.8.3 inferpy.prob_model module

1.8.4 inferpy.replicate module

1.8.5 inferpy.version module

1.8.6 Module contents