MXFusion

Release 1.0

Aug 31, 2018

Contents

1 Installation 3

2 Design Overview 5

3 API Reference 9

4 Tutorials! 11

5 Indices and tables 13

MXFusion is a library for integrating probabilistic modelling with deep learning.

MXFusion helps you rapidly build and test new methods at scale, by focusing on the modularity of probabilistic models and their integration with modern deep learning techniques.

CHAPTER 1

Installation

1.1 Dependencies / Prerequisites

MXFusion's primary dependencies are MXNet >= 1.2 and Networkx >= 2.1. See requirements.

1.2 Supported Architectures / Versions

MXFusion is tested on Python 3.5+ on MacOS and Amazon Linux.

1.3 pip

If you just want to use MXFusion and not modify the source, you can install through pip:

```
pip install mxfusion
```

1.4 From source

To install MXFusion from source, after cloning the repository run the following from the top-level directory:

```
pip install .
```

CHAPTER 2

Design Overview

2.1 Topical Guides

Working in MXFusion breaks up into two primary phases. Model definition involves defining the variables, distributions, and functions that make up your model. Inference then takes in real values and learns parameters for your model or gives predictions over the data.

2.1.1 Model Definition

MXFusion is a library for doing probabilistic modelling.

Probabilistic Models can be categorized into directed graphical models (DGM, Bayes Net) and undirected graphical models (UGM). Most popular probabilistic models are DGMs, so MXFusion currently only supports DGMs.

A DGM can be fully defined using 3 basic components: deterministic functions, probabilistic distributions, and random variables. As such, those are the primary ModelComponents in MXFusion.

Model

The primary data structure of MXFusion is the FactorGraph. FactorGraphs contain Variables and Factors. The FactorGraph exposes methods that Inference algorithms call such as drawing samples from the FactorGraph or computing the log pdf of a set of Variables contained in the FactorGraph.

When you want to start modelling, construct a Model object and start attaching ModelComponents to it. You can then see the Model's components by

```
m = Model()
m.v = Variable()
print(m.components)
```

When a ModelComponent is attached to a Model, it is automatically updated in the Model's internal data structures and will be included in any subsequent inference operations over the model.

Model Components

All ModelComponents in MXFusion are identified uniquely by a UUID.

###Variables In a model, there are typically four types of variables: a random variable following a probabilistic distribution, a variable which is the outcome of a deterministic function, a parameter (with no prior distribution), and a constant. The definitions of first two types of variables will be discussed later. The latter two types of variables can be defined with the following statement:

```
m.v = Variable(shape=(10,), constraint=PositiveTransformation())
```

At this stage, you do not need to specify whether v is a parameter or constant, because, if it is a constant, its value will be provided during inference, otherwise it will be treated as a parameter.

A typical example of when a constant would be specified at inference time is the size (shape) of an observed variable, which is known when data is provided. In the above example, we specify the name of the variable, the shape of the variable and the constraint that the variable has. It defines a 10-dimension vector whose values are always positive (v>=0).

Factors

####Distributions In a probabilistic model, random variables relate to each other through probabilistic distributions.

During model definition, the typical interface to generate a 2 dimensional random variable x from a zero mean unit variance Gaussian distribution looks like:

```
m.x = Normal.generate_variable(mean=0, variance=1, shape=(2,))
```

The two dimensions are independent to each other and both follow the same Gaussian distribution. The parameters or shape of a distribution can also be variables, for example:

```
m.mean = Variable(shape=(2,))
m.y_shape = Variable()
m.y = Normal.generate_variable(mean=m.mean, variance=1, shape=m.y_shape)
```

MXFusion also allows users to specify a prior distribution over pre-existing variables. This is particularly handy for interfacing with neural networks in MXNet because it allows you to set priors over parameters in an existing Gluon Block, such as a neural network implementation. The API for specifying a prior distribution looks like:

```
m.x = Variable(shape=(2,))
m.x.set_prior(Gaussian(mean=0, variance=1))
```

The above code defines a variable x and sets the prior distribution of each dimension of x to be a scalar unit Gaussian distribution.

Because Models are FactorGraphs, it is common to want to know what ModelComponents come before or after a particular component in the graph. These are accessed through the ModelComponent properties `successors` and `predecessors`.

```
m.mean = Variable()
m.var = Variable()
m.y = Normal.generate_variable(mean=m.mean, variance=m.var)
```

####Functions The last building block of probabilistic models are deterministic functions. The ability to define sophisticated functions allows users to build expressive models with a family of standard probabilistic distributions. As MXNet already provides full functionality for defining a function and automatically evaluating its gradients, Functions

in MXFusion are a wrapper over the functions in MXNet's Gluon interface. Functions are defined in standard MXNet syntax and provided to the MXFusion Function wrapper as below.

First we define a function in MXNet Gluon syntax using a Block object:

```
class F(mx.gluon.Block):
    def forward(self, x, y):
        return x*2+y
```

Then we create an MXFusion Function instance by passing in our Gluon function instance:

```
f_gluon = F()
m.f_mf = MXFusionGluonFunction(f_gluon)
```

Then this MXFusion Function can be called using MXFusion variables and its outcome will another variable[s] representing the outcome of the function:

```
m.x = Variable(shape=(2,))
m.y = Variable(shape=(2,))
m.f = f_mf(x, y)
```

FAQ

- Why don't you support undirected graphical models (UGM)?
- A UGM is typically defined in terms of a set of potential functions. Each potential function is a non-negative function that is defined on a subset of variables in a model. The joint probability distribution of an UGM is defined as the product of all the potential functions divided by a normalization term (known as a partition function).
- The notation of a DGM and an UGM can be unified into a factor graph, where a factor can be either a probabilistic distribution or a potential function. **In our implementation, the distribution UI is inherited from the factor abstract class, which enables future extension to support UGM** , although inference algorithms for UGM will be completely different.

2.1.2 Inference

Notes about inference in MXFusion.

Inference Algorithms

MXFusion currently supports stochastic variational inference.

Variational Inference

Variational inference is an approximate inference method that can serve as the inference method over generic models built in MXFusion. The main idea of variational inference is to approximate the (often intractable) posterior distribution of our model with a simpler parametric approximation, referred to as a variational posterior distribution. The goal is then to optimize the parameters of this variational posterior distribution to best approximate our true posterior distribution. This is typically done by minimizing the lower bound of the logarithm of the marginal distribution:

\begin{equation} \log p(y|z) = \log \int_x p(y|x) p(x|z) \geq \int_x q(x|y,z) \log \frac{p(y|x) p(x|z)}{q(x|y,z)} = \mathcal{L}(y,z), \label{eqn:lower_bound_1} \end{equation}

where $(y|x) p(x|z)$ forms a probabilistic model with x as a latent variable, $q(x|y)$ is the variational posterior distribution, and the lower bound is denoted as $\mathcal{L}(y,z)$. By then taking a natural exponentiation of $\mathcal{L}(y,z)$, we get a lower bound of the marginal probability denoted as $\tilde{p}(y|z) = e^{\mathcal{L}(y,z)}$.

A technical challenge with VI is that the integral of the lower bound of a probabilistic module with respect to external latent variables may not always be tractable. Stochastic variational inference (SVI) offers an approximated solution to this new intractability by applying Monte Carlo Integration. Monte Carlo Integration is applicable to generic probabilistic distributions and lower bounds as long as we are able to draw samples from the variational posterior.

In this case, the lower bound is approximated as \begin{equation} \mathcal{L}(l, z) \approx \frac{1}{N} \sum_i \log \frac{p(l|y_i)e^{\mathcal{L}(y_i,z)}}{q(y_i|z)}, \quad \mathcal{L}(y_i, z) \approx \frac{1}{M} \sum_j \log \frac{p(y_i|x_j) p(x_j|z)}{q(x_j|y_i, z)} , \end{equation} where $y_i|z \sim q(y|z)$, $x_j|y_i,z \sim q(x|y_i,z)$ and N is the number of samples of y and M is the number of samples of x given y. Note that if there is a closed form solution of $\tilde{p}(y_i|z)$, the calculation of $\mathcal{L}(y_i,z)$ can be replaced with the closed-form solution.

Let's look at a simple model and then see how we apply stochastic variational inference to it in practice using MXFusion.

Creating a Posterior

Examples

Saving Inference Results

2.2 Design Choices

2.2.1 CIPs

CIPs are a design proposal mechanism from the Apache Software Foundation.

CHAPTER 3

API Reference

mxfusion

CHAPTER 4

Tutorials!

Below is a list of tutorial / example notebooks demonstrating MXFusion's functionality.

4.1 Getting Started

4.2 Example Models

4.3 Developer Tutorials

CHAPTER 5

Indices and tables

- genindex
- modindex
- search