A Tutorial on the `distributions` Package

Samy Bengio
Dalle Molle Institute for Perceptual Artificial Intelligence (IDIAP)
CP 592, rue du Simplon 4,
1920, Martigny, Switzerland,
bengio@idiap.ch, http://www.idiap.ch/~bengio

October 2, 2002

1 Introduction

The `distributions` package is aimed at providing support for machine learning algorithms using the general and statistical concept of a distribution. A distribution is an object, such as a Gaussian, that can, for instance, compute a probability of a data, or the likelihood of a data set. The parameters of such a distribution can be trained using various training algorithms, such as gradient descent or expectation maximization. Moreover, the package also contains other related concepts such as conditional distributions.

2 Basic Concepts

As `GradientMachine` from which they inherit, `Distribution` objects usually operate on `Sequence` objects. For instance, one can ask what is the log likelihood of a sequence using the method `logProbability(Sequence* inputs)`. Note however that many distributions do not use the temporal aspect of a sequence and hence compute the log likelihood of a sequence as the sum of the log likelihood of the constituting frames of the sequence. Hence, for these distributions, the method to provide is `frameLogProbability(int t, real* f_inputs)`. For distributions that compute the log probabilities at the frame level (such as `DiagonalGMM`, the data member `log_probabilities` is a sequence of the same length as `inputs` which contains for each frame the value of the log probability of the frame (if the corresponding method has been called of course). For distributions that compute directly the log probability of the whole sequence (such as `HMM`, the data member `log_probabilities` is a sequence of length 1 which contains the log probability of the sequence.

As some distributions need to be initialized before being used on a given sequence, it is important to always call the method `eMSequenceInitialize(Sequence*`

`inputs)` or the method `sequenceInitialize(Sequence* inputs)` before using it (hence, if this is the case for your new distribution, don't forget to redefine such methods). The difference between the two methods corresponds to which training algorithm would eventually be used with this distribution.

3 Parameter Training

Numerous training algorithms can be applied in order to select the parameters of a given distribution. The most known are the gradient descent, the Expectation-Maximization (EM), and the Viterbi algorithms. For each of these algorithms, special methods have been created for each distributions and a specific `Trainer` object has also been provided. All these trainers should be able to provide the method `train(DataSet* data, MeasurerList *measurers)` to train a given distribution on a given dataset and provide measurements through a given list of measurers. They should also provide a method `test(MeasurerList *measurers)` that simply measures the performance of the new distribution over the datasets specified in the measurers.

3.1 EM Training

An introduction to the EM algorithm can be found in [1, 2]. Basically, it is an iterative and batch algorithm that loops through all the sequences in order to modify the parameters of a given distribution in order to maximize the likelihood of a given data set. The class `EMTrainer` implements such algorithm. The distribution methods to implement in order to use this type of training on a new distribution are the following:

- `setDataSet(data)`: if the distribution needs to be initialized using a given dataset, such as `DiagonalGMM` or `Kmeans` for instance.
- `eMIterInitialize()`: this method is called at the begining of every EM iteration. It is often used to set all the counters to 0.
- `eMForward(inputs)`: this method should normally not be modified. it calls the methods `eMSequenceInitialize(inputs)` and `logProbability(inputs)`.
- `eMSequenceInitialize(inputs)`: this method is called at the begining of every sequence. It is often used to precompute certain values which will be constant for the whole sequence.
- `logProbability(inputs)`: this method is used to compute the log probability of a whole sequence. If your distribution does not handle the time relations, then you probably don't need to modify it.
- `frameLogProbability(t, inputs)`: this method should return the log probability of a given frame of the inputs sequence.

- `eMAccPosteriors(inputs,log_posterior)`: this method should accumulate the log posterior of the hidden variables related to your distribution, weighted by the global log posterior given.
- `eMUpdate()`: this method modifies the parameters of the distribution, according to the log posteriors accumulated for each hidden variable of your distribution.

3.2 Viterbi Training

The Viterbi training algorithm is a simplified version of the EM algorithm where, instead of modifiying all the posteriors of all the hidden variables, only the most probable gets the whole credit. This training algorithm is implemented in the class `EMTrainer` using the option `viterbi` in the constructor. On top of the methods specialized for EM training, the following methods should also be defined:

- `viterbiForward(inputs)`: this method should normally not be modified. it calls the methods `eMSequenceInitialize(inputs)` and `viterbiLogProbability(inputs)`.
- `viterbiLogProbability(inputs)`: this method is used to compute the log probability of a whole sequence using the viterbi method. If your distribution does not handle the time relations, then you probably don't need to modify it.
- `viterbiFrameLogProbability(t, inputs)`: this method should return the log probability of a given frame of the inputs sequence, using the viterbi approximation.
- `viterbiAccPosteriors(inputs,log_posterior)`: this method should accumulate the log posterior of the most probable hidden variable related to your distribution given the input sequence.

3.3 Gradient Descent Training

Most distributions can also be trained using the more general gradient descent technique. For this to be possible, the distribution should be differentiable with respect to its parameters and a suitable criterion should be provided. A general criterion to minimize the negative log likelihood of a sequence (hence maximizing the likelihood) is provided and named `NLLCriterion`. The `Trainer` class that implements such training is `StochasticGradient`. Moreover the following methods of distribution should be modified:

- `setDataSet(data)`: if the distribution needs to be initialized using a given dataset, such as `DiagonalGMM` or `Kmeans` for instance.
- `iterInitialize()`: this method is called at the begining of every iteration. It is often used to set all the derivatives to 0.

- `forward(inputs)`: this method should normally not be modified. it calls the methods `sequenceInitialize(inputs)` and `logProbability(inputs)`.
- `sequenceInitialize(inputs)`: this method is called at the begining of every sequence. It is often used to precompute certain values which will be constant for the whole sequence.
- `logProbability(inputs)`: this method is used to compute the log probability of a whole sequence. If your distribution does not handle the time relations, then you probably don't need to modify it.
- `frameLogProbability(t, inputs)`: this method should return the log probability of a given frame of the inputs sequence.
- `backward(inputs,alpha)`: this method is used to compute the derivative of the parameters with respect to a given criterion. In fact the sequence `alpha` contains already the derivative with respect to the likelihood of the sequence `inputs`. If your distribution is not using any time relation, then it should normally not be modified.
- `frameBackward(f_inputs,f_alpha)`: this method is used to compute the derivative of the parameters with respect to a given criterion for a given frame only. In fact the frame vector `f_alpha` contains already the derivative with respect to the likelihood of the frame `f_inputs`.

Moreover, if one wants to optimize a different criterion than the `NLLCriterion`, then the new criterion (that should inherit from `Criterion`) should implement the following methods:

- `forward(inputs)`: should put in `outputs->frames[0][0]` the value of the criterion to optimize.
- `backward(inputs,alpha)`: should put in `beta->frames[0][0]` the gradient with respect to this criterion.

4 Examples of Distributions

In this section, I try to explain in more details some implementations regarding several typical distributions.

4.1 Multinomial

A `Multinomial` is a distribution over discrete events. It can estimate the probability of a random variable X to be in one of several finite state: $P(X = k)$ with $k \in \{0, 1, 2, \ldots, K-1\}$. Hence, the most important parameter in the constructor is the number of values K the random variable can take. For each of these values, a log probability will be kept (and eventually estimated over a dataset). These log probabilities are kept in the variable `log_weights`. In the following, you will find several details regarding this distribution:

- This distribution does not take time into account, of course.
- The size of the input vector given to its various methods is always equal to 1, and the input vector should contain an integer value that represents the various values the random variable can take, from 0 to `n_values` minus 1.
- The method `frameLogProbability(t, inputs)` returns the log probability of the random variable when it equals the integer value contained in `inputs`.
- The variable `weights_acc` contains the posterior of the random variable being equal to the corresponding value, which is used during EM training to estimate the log probability.
- The variable `prior_weights`, which is given in the constructor, is used to initialize the posterior during EM training in order to ensure that each value gets a minimum number of *observations*. This represents a Dirichlet prior on the distribution.
- The variable `dlog_weights` is used to keep the derivative of the criterion with respect to the parameters, when training with gradient descent.

4.2 DiagonalGMM

A `DiagonalGMM` is a distribution that represents a Gaussian Mixture Model with diagonal covariance matrix. The probability of a sequence X given such distribution is

$$p(\mathbf{X}) = \prod_{t=1}^{T} p(\mathbf{x}_t) = \prod_{t=1}^{T} \sum_{n=1}^{N} w_n \cdot \mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_n, \boldsymbol{\sigma}_n) \tag{1}$$

where $\mathcal{N}(\mathbf{x}_t; \boldsymbol{\mu}_n, \boldsymbol{\sigma}_n)$ is a Gaussian with mean $\boldsymbol{\mu}_n \in \mathbb{R}^d$ where d is the number of features and with $\boldsymbol{\sigma}_n$ the diagonal of the covariance matrix $\boldsymbol{\Sigma}_n \in \mathbb{R}^{d^2}$:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{1}{(2\pi)^{\frac{d}{2}} \sqrt{|\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right) \tag{2}$$

The following information is specific to such objects:

- This distribution does not take time into account.
- The size of the input vector given to its various methods is the same as the size of the mean and standard deviation vectors.
- The number of Gaussians is kept in `n_gaussians`.
- The parameters of this distributions are:

  - `log_weights`: a vector containing the log of the prior probability of each Gaussian.
  - `means`: a matrix of `n_gaussians` vectors of means.
  - `var`: a matrix of `n_gaussians` vectors of variances.

- A minimum variance for each dimension can be specified with the method `setVarThreshold` and is kept in the variable `var_threshold`.

- When the goal is to train such distribution using EM, the initialization procedure is very important. The most used solution for this is to use a Kmeans algorithm, providing that an `initial_kmeans_trainer` is given to the constructor. An optional `Measurer` can also be provided to measure the Kmeans training.

- While it is possible to train a `DiagonalGMM` by gradient descent, it is very sensitive to initial condition and optimization problems are to be expected regarding the variance.

- For each frame of the current sequence and each Gaussian, the log probability of the frame given the Gaussian is kept in the `Sequence` variable `log_probabilities_g`.

- In order to accelerate some computation, methods `eMIterInitialize` and `sequenceInitialize` both recompute several intermediate variables that are needed to compute the likelihood. One of these methods should imperatively be called before computing a likelihood.

4.3 HMM

An `HMM` is a distribution that represents Hidden Markov Models (HMMs). A good introduction to HMMs can be found in [3]. Basically, an HMM can model the density $P(X)$ of sequences X using a factored representation based on a set of states which are represented by emission distributions $P(X|q_t = i)$, where q_t is the state at time t, and a table of transition probabilities $P(q_t = i|q_{t-1} = j)$. It can be trained by EM, Viterbi, or Gradient Descent, depending on the kind of distributions used in the states. Moreover, the following information is specific to HMMs:

- The number of states of the HMM is kept in `n_states`. Note however that an HMM always contain an initial and a final states which are non emitting. Hence `n_states` is 2 plus the effective number of states.

- The emission distributions are kept in the table `states`. Note that `states[0]` and `states[n_states-1]` are `NULL`.

- The transitions are kept in log only and are to be found in the matrix `log_transitions`, which is `n_states` times `n_states`. It is initialized using the variable `transitions` which is given in the constructor. Only

the non zero transitions will be considered, hence the non zero transitions represent the transition distributions.

- In order to train such model, the well-known *forward-backward* procedure is used, with the following variables:
  - `log_alpha`: contains the log of the alpha variable. During Viterbi training, it contains the Viterbi score instead.
  - `log_beta`: contains the log of the beta variable.
  - `log_probabilities_s`: contains for each non-null state and each frame of the current sequence, the log probability of the frame given the state.
- The following variables are used during EM or Viterbi training:
  - `transitions_acc` which is a matrix of the posteriors of each transition.
  - `prior_transitions` is used to initialize the posteriors on the transitions with a Dirichlet prior.
  - The initialization of the model is done using either a linear segmentation (assuming a left-to-right model, each sequence is segmented linearly along the states and each state is then initialized using its assigned frames) or a full segmentation (each state is assigned a random selection of frames and then initialized using these frames).
- The following variables are used during gradient descent training:
  - `dlog_transitions` which keeps the derivative of the criterion with respect to each transition.
- During Viterbi decoding (with the method `decode`), the variable `arg_viterbi` keeps track of the previous state in the optimal sequence. Moreover, the `Sequence` variable `viterbi_sequence` contains the optimal sequence of states after decoding.
- The method `printTransitions` can be used (during debug for instance) to print the current value of the transition table.

References

[1] C. Bishop. *Neural Networks for Pattern Recognition*. Clarendon Press, Oxford, 1995.

[2] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum-likelihood from incomplete data via the EM algorithm. *Journal of Royal Statistical Society B*, 39:1–38, 1977.

[3] Laurence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.