# distributed Structured Prediction
# -
# Documentation

A. G. Schwing,
T. Hazan,
M. Pollefeys,
R. Urtasun

July 2013

# Contents

# Abstract

*distributed Structured Prediction* (dSP) is the most general publicly available implementation of algorithms for learning and inference in graphical models. On top of that it provides multiple mechanisms for parallelism to best leverage high performance computing environments as well as single computers. The key features of this software package are:

- **Support for higher order regions:** Arbitrary Hasse diagrams are taken as input for both learning and inference (Yedidia *et al.*, 2005; Hazan *et al.*, 2012). Therefore dSP supports junction tree optimization just like ordinary message passing on a factor graph. In addition and at the expense of exponential complexity, models can be manually tightened up to the tree-width.

- **Support for arbitrary counting numbers:** The user is free to specify the counting numbers. Therefore, non-convex Bethe approximations (Pearl, 1988) are supported just like convex Belief Propagation (Weiss *et al.*, 2007; Meltzer *et al.*, 2009; Hazan & Shashua, 2008, 2010). Parameters determine whether updates for the Lagrange multipliers are interleaved with gradient steps for the model parameters (Hazan & Urtasun, 2010).

- **Suitable for hidden conditional random fields or structured support vector machines:** Changing the parameter $\epsilon$ allows optimization of the structured support vector machines (SSVM) or max-margin markov network ($M^3N$) objective (max-margin formulation) (Taskar *et al.*, 2003; Tsochantaridis *et al.*, 2005) for $\epsilon = 0$ just like the conditional random field (CRF) (Lafferty *et al.*, 2001) cost function (max-likelihood) for $\epsilon = 1$.

- **Support for high performance computing:** The package contains modules that parallelize inference w.r.t. samples on a single machine or w.r.t. the graph coloring of the Hasse diagram (Schwing *et al.*, 2011). In addition, learning for distributed memory environments supports parallelization by dividing the model or by dividing the samples (Schwing *et al.*, 2012b). Similarly, parameters determine whether inference processes multiple samples concurrently or whether the model is partitioned onto multiple computers. To suit large clusters we utilize standard message passing libraries like OpenMPI or MPICH. To save time for the transmission, we also merge messages of different regions into a single package.

- **Different input formats:** We designed both a textual as well as a binary input file format. While the first is suitable for debugging purposes, the latter targets large scale data sets. Besides file formats a Matlab mex function provides direct access to the modules.

- **Latent variable models:** We provide a module to learn from weakly labeled data, *i.e.*, we support latent SSVMs (Yu & Joachims, 2009) just like hidden CRFs (Quattoni *et al.*, 2007) by following (Schwing *et al.*, 2012a). Parallelization w.r.t. samples and by graph coloring is also supported for latent variable models.

At the moment we refer the interested reader to respective publications for theoretical discussions and mathematical derivations. The purpose of this document is to provide an introduction to start setting up your own models using *distributed Structured Prediction* (dSP).

# 1 Introduction

The *distributed Structured Prediction* (dSP) software package provides an interface to learning with graphical models. No detailed understanding of the underlying algorithms is required and we therefore focus on the important aspects only.

The current version of this tool has quite a few predecessors, some of which were made available to a restricted set of users. Besides our own intensive usage for computer vision applications such as () we also got valuable feedback from projects such as (). This helped to revise the internal data structures and interfaces multiple times resulting in what we now refer to as version 5.0. We hope to provide a nice user experience but we are always looking forward to your feedback and suggestions.

# 2 Technical Introduction

Inference and learning are the two tasks commonly addressed when working with graphical models.

## 2.1 Learning

Let $x \in \mathcal{X}$ denote an object from the input space, *e.g.*, an image or a sentence, while $s \in \mathcal{S}$ refers to discrete output space objects, *e.g.*, segmentations or parse trees. For learning we are given a data set of samples $(x, s) \in \mathcal{D} = \{(x_i, s_i)\}_i$. We let $\phi(x, s) \in \mathbb{R}^F$ map from the input-output product space to an $F$-dimensional measurement vector, also referred to as feature vector.

In addition we construct a loss function $\ell(\hat{s}, s)$ which measures the fitness of an output space estimate $\hat{s}$ with the provided ground truth annotation $s$. Commonly we like output space objects that differ largely from the ground truth to have a high loss.

The learning task optimizes the following program with $w \in \mathbb{R}^F$ denoting a weight vector which linearly combines the measurements to obtain a discriminative function for the considered task:

$$\min_w \frac{C}{p}\|w\|_p^p + \sum_{(x,s)} \epsilon \ln \sum_{\hat{s}} \exp \frac{w^\top \phi(x, \hat{s}) + \ell_s(\hat{s})}{\epsilon} - w^\top \sum_{(x,s)} \phi(x, s). \quad (2.1)$$

Note the summation over all output space configurations $\hat{s}$ which makes this task intractable in general.

For many applications the output space contains multiple variables of interest, *e.g.*, all the individual pixel labels or the part of speech for every word. Let us refer to a subset of variables of the output space via $s_r$,

with region $r$ being an arbitrary sized set of variable indexes. Using this notation we are able to refer to a single variable within the output space just like we are able to subsume all the variables within one set. The more local the measurements the easier the learning and inference tasks. Hence our interest to keep the size of the regions low.

Directly influenced by the locality is the description of the features $\phi(x, s)$ which is the important part for every user. The measurements are the key ingredients responsible for the success of the application. Every element $\phi_k(x, s)$ $k \in \{1, \ldots, F\}$ of the feature vector is assumed to be decomposable into a set of functions with local dependencies only, *i.e.*,

$$\phi_k(x, s) = \sum_{r \in \mathcal{R}_k} \phi_{k,r}(x, s_r). \tag{2.2}$$

The set $\mathcal{R}_k$ subsumes all the regions important for the $k$-th feature. The set of all regions is given by $\mathcal{R} = \bigcup_{k \in \{1,\ldots,F\}} \mathcal{R}_k$ and we assign a counting number $c_r$ to every region.

## 2.2 Inference

During inference we employ the weight vector $w$ to compute potentials $\theta_r(s_r) = \sum_{k:r \in \mathcal{R}_k} w_k \phi_{k,r}(x, s_r)$. Hence an approximate inference task considers

$$\max_b \sum_{r, s_r} \theta_r(s_r) b_r(s_r) + \epsilon c_r H(b_r) \tag{2.3}$$

where $b_r$ is the set of local beliefs over the region $r \in \mathcal{R}$ rather than a single belief over the full output space. The entropy is referred to via $H$. Also note the summation over the configurations of local regions $s_r$ only which underlines the importance of finding local parameterizations for the considered application.

# 3 The Matlab interface

## 3.1 Textual Interface

All the training samples are subsumed within one cell array of structures, *i.e.*,

```
Samples{x}
```

where `x` is the index of the sample. For every sample we specify the cardinality of all the variables within the output space via an array, *e.g.*,

```
Samples{x}.VariableCardinalities = [2 2 2 2 2 2 2 2];
```

In addition we assign every variable to a specific computer index (starts counting from 1) via

```
Samples{x}.MachineIDs = [1 1 1 1 2 2 2 2];
```

This is not important when using *dSP* from within matlab and also not important when using the framework locally. However it allows to distribute inference and learning onto multiple computers when considering high performance compute environments.

For learning it is also important to provide observations/annotations. This is achieved via

```
Samples{x}.Observation = [1 1 2 2 2 2 1 1];
```

where we assign a specific state to every variable. The first state is referred to by 1 and the specific state chosen for a variable is obviously less or equal to its cardinality. An observation equal zero means that this variable is latent/hidden.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 |
| $x_2$ | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 |
| Element: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

Table 3.1: Ordering within the loss and feature vectors for two variables within a region.

We already emphasized the importance of having a local representation. This is reflected within the matlab data structure via the cell array of regions, *i.e.*,

```
Samples{x}.Regions{r}
```

Some of its fields are the counting number and the variable indices involved in that region.

```
Samples{x}.Regions{r}.c_r = 1;
Samples{x}.Regions{r}.VariableIndices = [k];
```

In addition every region is assigned a loss function $\ell(\hat{s}_r)$ which measures the compatibility of the estimate $\hat{s}_r$ with the samples annotation. The loss function is specified via the array

```
Samples{x}.Regions{r}.Loss = [0 0];
```

Note that the ordering within that array is important. The first element corresponds to all the variables specified within the variable indices array to take the first state. The second entry corresponds to the first variable to take state 2 while the other ones remain at state 1, *etc.* The ordering is also illustrated in Tab. 3.1.

In addition regions may not have parents assigned

```
Samples{x}.Regions{r}.Parents = [];
```

or it may be connected to other regions having the variables within region $r$ as a subset:

```
Samples{x}.Regions{r}.Parents = [10 11 13]
```

This parent relation ship describes the Hasse diagram, a generalization of a factor graph, and is used to enforce marginalization constraints between the local beliefs $b_r$.

Arguably the most important part for each region are its features. Every region is involved in one or more features $\phi_{r,k}(s_r)$ subsumed within the cell array

```
Samples{1}.Regions{r}.Features{ix}
```

To express the weight vector it is multiplied with we use the field `r`, *i.e.*,

```
Samples{1}.Regions{r}.Features{ix}.r = k;
```

which starts counting from 1. The respective potential is specified via

```
Samples{1}.Regions{r}.Features{ix}.pot = [1 -1];
```

and follows the same ordering as the loss, *i.e.*, the ordering specified within Tab. 3.1.

Having specified all the samples in the above illustrated way we employ the function

```
WriteOutput(Samples);
```

to write the feature and observation files onto the hard disk. The resulting files can be used as input when working in environments that don't provide access to matlab. Note that the resulting files are text files and merely intended for debugging purposes. For large scale applications it is certainly more suitable to utilize the binary interface detailed below.

To call *dSP* we utilize either

```
[w, Prediction] = structuredPrediction(Samples, Params, task, w_init);
```

or

```
[w, Prediction] = latentStructuredPrediction(Samples, Params, task, w_init
```

The learning and inference parameters, detailed below, are set via the structure `Params`, while the variable task indicates to do learning (`task < 2`) or inference (`task > 0`). Note that `task = 1` allows learning and inference right after each other in order to avoid the computational overhead for constructing internal data structures. The weight vector used to initialize training or employed during inference is specified via the $F$-dimensional array `w_init`. All variables besides `Samples` are optional with default `task = 0` and `w_init` being an $F$-dimensional weight vector of all ones.

## 3.2 Binary Interace

The binary interface is more suitable for large scale applications. At the moment it is suitable if features are stored on the hard disk and loaded into the standalone executables by specifying the `binary` option. The format is very similar to the textual input with a few minor modifications being described subsequently.

Instead of specifying the potential of every region and every feature vector element separately, we share potentials by just providing a potential index rather than the potential itself, *i.e.*,

```
Samples{x}.Regions{r}.Features{ix}.potIX = k;
```

Similarly we just provide the potential index of the loss via

```
Samples{x}.Regions{r}.LossIX = someIX;
```

rather than specifying the loss directly.

The potential values are then given by a specific cell array

```
Samples{x}.Potentials{ix}
```

with each entry having the two fields

```
Samples{1}.Potentials{anything}.IX = 3;
Samples{1}.Potentials{anything}.pot = zeros(1,dimensions);
```

The field `IX` refers to the index of the potential used within either `potIX` or `LossIX` or both, while the field `pot` defines the potential itself.

To store the features on the hard drive in a binary file format the function

```
WriteOutputBinary(Samples);
```

is used. In addition

```
WriteModel(w_init);
```

is available to construct a binary model file to be used as an initializer.

We also provide a function

```
Samples = ReadOutput(fileName);
```

which takes the binary output produced from a standalone executable and converts it back into a matlab format.

For large scale examples we hence suggest the following workflow:

1. Compute or features within Matlab or directly within C++ and store them on the hard disk.
2. Process the data using one of the provided standalone executables (see README for details).
3. Evaluate or continue processing the result either in matlab or via specific C++ programs.

# 4 Feature Files

Instead of generating the features using the provided Matlab functionality outlined above, it is naturally possible to use your custom programs to generate equivalent files. For this purpose we provide a short description of the file formats having extension `.feature` and `.observation`. Note that every sample has its own file which is useful if the task is parallelized onto multiple machines w.r.t. samples, since no blocking access occurs.

## 4.1 Textual Format

As a relict from the UAI file format[1] every textual file format starts with the identifier `MARKOV`. In the next line we store the number of variables involved in the sample. The next line contains the cardinality of all the variables separated by space characters. The subsequent line contains the IDs for the machines that a variable is assigned to when distributing the models in a more fine grained manner than distributing per sample. The IDs are null-based, *i.e.*, we start counting with zero.

Every of the subsequent lines specified a single region $r$. The parameters of every region are separated by space characters. The first integer is a region index followed by a textual float number for the counting number $c_r$. The next integer specifies the number of variables involved in that region before we list all the variable indices (starting from zero for the first). Afterwards we indicate in how many features this region is involved before listing for every feature the weight vector index (zero based) and the corresponding potential. Note that the size of the potential is defined by the involved variables and the ordering follows the matlab format, *i.e.*, Tab. 3.1. After having listed all the features we store the loss potential

[1] http://www.cs.huji.ac.il/project/PASCAL/fileFormat.php

before providing the number of parents that this region is connected to and the region indices of the parents.

Every region line can contain a comment at the end containing at most 100 characters. The tag to start the comment is `#`.

The observation file contains in its first line the number of variables of this sample. The subsequent line contains the observed state of every variable. The first state is referred to via 0 and no observation is indicated by $-1$.

## 4.2 Binary Format

The binary file starts off with a 32-bit integer providing the number of variables involved in this sample. The next array of 32-bit integers of size corresponding to the number of variables provides their cardinalities while a subsequent 32-bit integer array of equal length denotes the machine assignments.

The next 32-bit integer provides the number of regions specified within this file. For every region we then store

- the region id as a 32-bit integer
- the counting number $c_r$ as a 64-bit double
- the number of variables involved and their indices (zero based) as 32-bit integers
- the number of feature elements and for every feature element the zero-based weight vector index as well as the potential index as 32-bit integers
- the potential index of the loss function as 32-bit integer
- the number of parents and the parent region indices also as 32-bit integers

The next 32-bit integer provides the number of potentials. For every potential we then store

- the potential index as 32-bit integer
- the size of the potential as a 32-bit integer
- the potential entries as 64-bit doubles

The observation file is a textual file as described in the previous section.

# 5 Parameters

Subsequently we describe the parameters to adjust the algorithms within the *dSP* framework in alphabetic order. The parameters are given to the program via "`<flag> <NumberOrString>`"

- **(-a) ArmijoIterations:** how many Armijo iterations to perform.
- **(-b) BetheCountingNumbers:** whether to use bethe counting numbers for the latent variable model. The counting numbers for structured prediction are always specified within the input files.
- **(-c) C:** as given in Eq. (2.1).
- **(-i) CCCPIterations:** how many CCCP iterations to perform when training with latent variables.
- **(-f) CRFEraseMessages:** whether to erase the messages after every iteration. Useful if you want to learn CRFs or structured Support Vector Machines the traditional way.
- **(-j) CRFIterations:** how many CRF iterations during one CCCP iteration.
- **(-k) CRFMPIterations:** how many message passing iterations during one CRF iteration.
- **(-o) CRFOuterExchange:** every how many iterations should messages between computers be exchanged if the model is divided onto multiple machines
- **(-x) CRFOuterIterations:** how often to exchange messages between computers for computation of the gradient or inference when distributing the model onto multiple machines

- **(-e) epsilon:** as given in Eq. (2.1) and Eq. (2.3).
- **(-m) MPIterations:** how many message passing iterations during inference.
- **(-p) p:** as given in Eq. (2.1).
- **(-y) ReadBinary:** use the binary input format
- **(-r) ReuseMessagesForF:** whether to reuse the previously computed messages during the Armijo iterations or whether to go with the traditional approach, *i.e.*, perform inference again
- **(-v) Verbosity:** more or less debug output

In addition a few program specific parameters are available

- **(-d) directory:** which directory to get the data from
- **(-t) task:** which task to execute 0: just learning, 1: learning and inference, 2: just inference
- **(-w) weight vector file:** file that contains information on how to initialize the weight vector
- **(-l) leave file:** output file, *i.e.*, which file to store the result in (weight vector and prediction result)

# 6 Conclusions

Have fun, enjoy and please let us know about bugs or suggestions on how to improve the framework. Please browse to `http://alexander-schwing.de` for further information and updates.

# Bibliography

Hazan, T. and Shashua, A. Convergent message-passing algorithms for inference over general graphs with convex free energy. In *Proc. UAI*, 2008.

Hazan, T. and Shashua, A. Norm-Product Belief Propagation: Primal-Dual Message-Passing for LP-Relaxation and Approximate-Inference. *Trans. on Information Theory*, 2010.

Hazan, T. and Urtasun, R. A Primal-Dual Message-Passing Algorithm for Approximated Large Scale Structured Prediction. In *Proc. NIPS*, 2010.

Hazan, T., Peng, J., and Shashua, A. Tightening Fractional Covering Upper Bounds on the Partition Function for High-Order Region Graphs. In *Proc. UAI*, 2012.

Lafferty, J., McCallum, A., and Pereira, F. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proc. ICML*, 2001.

Meltzer, T., Globerson, A., and Weiss, Y. Convergent message passing algorithms - a unifying view. In *Proc. UAI*, 2009.

Pearl, J. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers, 1988.

Quattoni, A., Wang, S., Morency, L.-P., Collins, M., and Darrell, T. Hidden-state Conditional Random Fields. *PAMI*, 2007.

Schwing, A. G., Hazan, T., Pollefeys, M., and Urtasun, R. Distributed Message Passing for Large Scale Graphical Models. In *Proc. CVPR*, 2011.

Schwing, A. G., Hazan, T., Pollefeys, M., and Urtasun, R. Efficient Structured Prediction with Latent Variables for General Graphical Models. In *Proc. ICML*, 2012a.

Schwing, A. G., Hazan, T., Pollefeys, M., and Urtasun, R. Distributed Structured Prediction for Big Data. In *Proc. NIPS Workshop on Big Learning*, 2012b.

Taskar, B., Guestrin, C., and Koller, D. Max-Margin Markov Networks. In *Proc. NIPS*, 2003.

Tsochantaridis, I., Joachims, T., Hofmann, T., and Altun, Y. Large Margin Methods for Structured and Interdependent Output Variables. *JMLR*, 2005.

Weiss, Y., Yanover, C., and Meltzer, T. MAP Estimation, Linear Programming and Belief Propagation with Convex Free Energies. In *Proc. UAI*, 2007.

Yedidia, J. S., Freeman, W. T., and Weiss, Y. Constructing free-energy approximations and generalized belief propagation algorithms. *Trans. on Information Theory*, 2005.

Yu, C.-N. and Joachims, T. Learning Structural SVMs with Latent Variables. In *Proc. ICML*, 2009.