Torch-Struct:
Deep Structured Prediction Library

Alexander M. Rush
Cornell Tech
Department of Computer Science
arush@cornell.edu

Abstract

The literature on structured prediction for NLP describes a rich collection of distributions and algorithms over sequences, segmentations, alignments, and trees; however, these algorithms are difficult to utilize in deep learning frameworks. We introduce Torch-Struct, a library for structured prediction designed to take advantage of and integrate with vectorized, auto-differentiation based frameworks. Torch-Struct includes a broad collection of probabilistic structures accessed through a simple and flexible distribution-based API that connects to any deep learning model. The library utilizes batched, vectorized operations and exploits auto-differentiation to produce readable, fast, and testable code. Internally, we also include a number of general-purpose optimizations to provide cross-algorithm efficiency. Experiments show significant performance gains over fast baselines. Case studies demonstrate the benefits of the library. Torch-Struct is available at https://github.com/harvardnlp/pytorch-struct.

1 Introduction

Structured prediction is an area of machine learning focusing on representations of spaces with combinatorial structure, as well as algorithms for inference and parameter estimation over these structures. Core methods include both tractable exact approaches like dynamic programming and spanning tree algorithms as well as heuristic techniques such linear programming relaxations and greedy search.

Structured prediction has played a key role in the history of natural language processing. Example methods include techniques for sequence labeling and segmentation (Lafferty et al., 2001; Sarawagi and Cohen, 2005), discriminative dependency and constituency parsing (Finkel et al., 2008; McDonald et al., 2005), unsupervised learning for labeling and alignment (Vogel et al., 1996; Goldwater and Griffiths, 2007), approximate translation decoding with beam search (Tillmann and Ney, 2003), among many others.

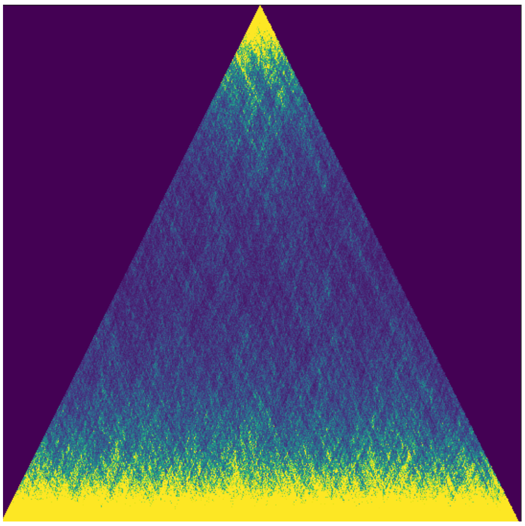


Figure 1: Distribution of binary trees over an 1000-token sequence. Coloring shows the marginal probabilities of every span. Torch-Struct is an optimized collection of common CRF distributions used in NLP that is designed to integrate with deep learning frameworks.

In recent years, research into deep structured prediction has studied how these approaches can be integrated with neural networks and pretrained models. One line of work has utilized structured prediction as the final layer for deep models (Collobert et al., 2011; Durrett and Klein, 2015). Another has incorporated structured prediction within deep learning models, exploring novel models for latent-structure learning, unsupervised learning, or model control (Johnson et al., 2016; Yogatama et al., 2016; Wiseman et al., 2018). We aspire to make both of these use-cases as easy to use as standard neural networks.

The practical challenge of employing structured

Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 335–342
July 5 - July 10, 2020. ©2020 Association for Computational Linguistics

Name	Structure ($\mathcal{Z}$)	Parts ($\mathcal{P}$)	Algorithm ($A(\ell)$)	LoC	T/S	Sample Reference
Linear-Chain, HMM	Labeled Chain	Edges (NC^2)	Forward-Backward	20	390k	(Lafferty et al., 2001)
Factorial-HMM	Labeled Chains	Trans. (LC^2) Obs. (NC^L)	Factorial F-B	20	25k	(Ghahramani and Jordan, 1996)
Alignment	Alignment	Match (NM) Skips ($2NM$)	DTW, CTC	50	13k	(Needleman and Wunsch, 1970)
Semi-Markov	Seg. Labels	Edge(NKC^2)	Segmental F-B	30	87k	(Baum and Petrie, 1966) (Sarawagi and Cohen, 2005)
Context-Free	Labeled Tree	CF Rules (G) Term. (CN)	I-O CKY	70	37k	(Kasami, 1966)
Simple CKY	Labeled Tree	Splits (CN^2)	0-th order CKY	30	118k	(Kasami, 1966)
Dependency	Proj. Tree	Arcs (N^2)	Eisner Alg	40	28k	(Eisner, 2000)
Dep (NP)	Non-Proj. Tree	Arcs (N^2)	Matrix-Tree Chiu-Liu (MAP)	40	1.1m	(Koo et al., 2007) (McDonald et al., 2005)
Auto-Regressive	Sequence	Prefix (C^N)	Greedy Search, Beam Search	60	-	(Tillmann and Ney, 2003)

Table 1: Models and algorithms implemented in Torch-Struct. Notation is developed in Section 5. *Parts* are described in terms of sequence lengths N, M, label size C, segment length K, and layers / grammar size L, G. Lines of code (*LoC*) is from the log-partition ($A(\ell)$) implementation. *T/S* is the tokens per second of a batched computation, computed with batch 32, $N = 25, C = 20, K = 5, L = 3$ (K80 GPU run on Google Colab).

prediction is that many required algorithms are difficult to implement efficiently and correctly. Most projects reimplement custom versions of standard algorithms or focus particularly on a single well-defined model class. This research style makes it difficult to combine and try out new approaches, a problem that has compounded with the complexity of research in deep structured prediction.

With this challenge in mind, we introduce Torch-Struct with three specific contributions:

- *Modularity*: models are represented as distributions with a standard flexible API integrated into a deep learning framework.
- *Completeness*: a broad array of classical algorithms are implemented and new models can easily be added.
- *Efficiency*: implementations target computational/memory efficiency for GPUs and the backend includes extensions for optimization.

In this system description, we first motivate the approach taken by the library, then present a technical description of the methods used, and finally present several example use cases.

2 Related Work

Several software libraries target structured prediction. Optimization tools, such as SVM-struct (Joachims, 2008), focus on parameter estimation. Model libraries, such as CRFSuite (Okazaki, 2007), CRF++ (Kudo, 2005), or NCRF++(Yang and Zhang, 2018), implement inference for a fixed set of popular models, usually linear-chain CRFs. General-purpose inference libraries, such as PyStruct (Müller and Behnke, 2014) or TurboParser (Martins et al., 2010), utilize external solvers for (primarily MAP) inference such as integer linear programming solvers and ADMM. Probabilistic programming languages, for example languages that integrate with deep learning such as Pyro (Bingham et al., 2019), allow for specification and inference over some discrete domains. Most ambitiously, inference libraries such as Dyna (Eisner et al., 2004) allow for declarative specifications of dynamic programming algorithms to support inference for generic algorithms. Torch-Struct takes a different approach and integrates a library of optimized structured distributions into a vectorized deep learning system. We begin by motivating this approach with a case study.

3 Motivating Case Study

While structured prediction is traditionally presented at the output layer, recent applications have deployed structured models broadly within neural networks (Johnson et al., 2016; Kim et al., 2017; Yogatama et al., 2016, inter alia). Torch-Struct aims to encourage this general use case.

To illustrate, we consider a latent tree model. ListOps (Nangia and Bowman, 2018) is a dataset of mathematical functions. Each input/output pair consists of a prefix expression x and its result y, e.g.

$$x = [\text{ MAX } 2\ 9\ [\text{ MIN } 4\ 7\ ]\ 0\ ] \quad y = 9$$

Models such as a flat RNN will fail to capture the hierarchical structure of this task. However, if a model can induce an explicit latent z, the parse tree of the expression, then the task is easy to learn by a tree-RNN model $p(y|x,z)$ (Yogatama et al., 2016; Havrylov et al., 2019).

Let us briefly summarize a latent-tree RL model for this task. The objective is to maximize the probability of the correct prediction under the expectation of a prior tree model, $p(z|x;\phi)$,

$$Obj = \mathbb{E}_{z\sim p(z|x;\phi)}[\log p(y \mid z, x)]$$

Computing the expectation is intractable so policy gradient is used. First a tree is sampled $\tilde{z} \sim p(z|x;\phi)$, then the gradient with respect to ϕ is approximated as,

$$\frac{\partial}{\partial \phi} Obj \approx (\log p(y \mid \tilde{z}, x) - b)(\frac{\partial}{\partial \phi} p(z|x;\phi))$$

where b is a variance reduction baseline. A common choice is the self-critical baseline (Rennie et al., 2017),

$$b = \log p(y \mid z^*, x) \text{ with } z^* = \arg\max_z p(z|x;\phi)$$

Finally an entropy regularization term is added to the objective encourage exploration of different trees, $Obj + \lambda\mathbb{H}(p(z \mid x;\phi))$.

Even in this brief overview, we can see how complex a latent structured learning problem can be. To compute these terms, we need 5 different properties of the structured prior model $p(z \mid x;\phi)$:

Sampling Policy gradient, $\tilde{z} \sim p(z \mid x;\phi)$

Density Score policy samples, $p(z \mid x;\phi)$

Gradient Backpropagation, $\frac{\partial}{\partial \phi} p(z \mid x;\phi)$

Argmax Self-critical, $\arg\max_z p(z \mid x;\phi)$

Entropy Objective regularizer, $\mathbb{H}(p(z \mid x;\phi))$

For structured models, each of these terms is non-trivial to compute. A goal of Torch-Struct is to make it seamless to deploy structured models for these complex settings.

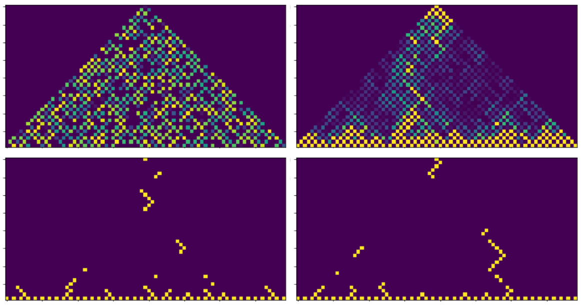


Figure 2: Latent Tree CRF example where each cell represents a span (i,j). Torch-Struct can be used to compute many different properties of a structured distribution. (a) Log-potentials ℓ for each part/span. (b) Marginals for CRF(ℓ) computed by backpropagation. (c) A single argmax tree $\arg\max_z$ CRF($z;\ell$). (d) A single sampled tree $z \sim$ CRF(ℓ).

4 Library Design

The library design of Torch-Struct follows the distributions API used by both TensorFlow and PyTorch (Dillon et al., 2017). For each structured model in the library, we define a conditional random field (CRF) distribution object. From a user's standpoint, this object provides all necessary distributional properties. Given log-potentials ℓ output from a deep network, the user can request samples $z \sim$ CRF(ℓ), probabilities CRF($z;\ell$), modes $\arg\max_z$ CRF(ℓ), or other distributional properties such as $\mathbb{H}$(CRF(ℓ)). The library is agnostic to how these are utilized, and when possible, they allow for backpropagation to update the input network. The same distributional object can be used for standard output prediction as for more complex operations like attention or reinforcement learning.

Figure 2 demonstrates this API for a binary tree CRF over an ordered sequence, such as $p(z \mid x;\phi)$ from the previous section. The distribution takes in log-potentials ℓ which score each possible span in the input. The distribution converts these to probabilities of a specific tree. This distribution can be queried for predicting over the set of trees, sampling a tree for model structure, or even computing entropy over all trees.

Table 1 shows all of the structures and distributions implemented in Torch-Struct. While each is internally implemented using different specialized algorithms and optimizations, from the user's perspective they all utilize the same external distributional API, and pass a generic set of distributional tests.[1] This approach hides the internal complexity

[1] The test suite for each distribution enumerates over all

of the inference procedure, while giving the user full access to the model.

5 Technical Approach

5.1 Conditional Random Fields

We now describe the technical approach underlying the library. To establish notation, first consider the implementation of a softmax categorical distribution, $\textsc{Cat}(\ell)$, with one-hot categories z with $z_i = 1$ from a set $\mathcal{Z}$ and probabilities given by the softmax over logits ℓ,

$$\textsc{Cat}(z;\ell) = \frac{\exp(z \cdot \ell)}{\sum_{z' \in \mathcal{Z}} \exp(z' \cdot \ell)} = \frac{\exp \ell_i}{\sum_{j=1}^{K} \exp \ell_j}$$

Define the log-partition as $A(\ell) = \mathrm{LSE}(\ell)$, i.e. log of the denominator, where LSE is the log-sum-exp operator. Computing probabilities or sampling from this distribution, requires enumerating $\mathcal{Z}$ to compute the log-partition A. A useful identity is that derivatives of A yield category probabilities,

$$p(z_i = 1) = \frac{\exp \ell_i}{\sum_{j=1}^{n} \exp \ell_j} = \frac{\partial}{\partial \ell_i} A(\ell)$$

Other distributional properties can be similarly extracted from variants of the log-partition. For instance, define $A^*(\ell) = \log \max_{j=1}^{K} \exp \ell_j$ then[2]: $\mathbb{I}(z_i^* = 1) = \frac{\partial}{\partial \ell_i} A^*(\ell)$.

Conditional random fields, $\mathrm{CRF}(\ell)$, extend the softmax to combinatorial spaces where $\mathcal{Z}$ is exponentially sized. Each z, is now represented as a binary vector over polynomial-sized set of *parts*, $\mathcal{P}$, i.e. $\mathcal{Z} \subset \{0,1\}^{|\mathcal{P}|}$. Similarly log-potentials are now defined over parts $\ell \in \mathbb{R}^{|\mathcal{P}|}$. For instance, in Figure 2 each span is a part and the ℓ vector is shown in the top-left figure. Define the probability of a structure z as,

$$\mathrm{CRF}(z;\ell) = \frac{\exp z \cdot \ell}{\sum_{z'} \exp z' \cdot \ell} = \frac{\exp \sum_p \ell_p z_p}{\sum_{z'} \exp \sum_p \ell_p z'_p}$$

Computing probabilities or sampling from this distribution, requires computing the log-partition term A. In general, computing this term is now intractable, however for many core algorithms in NLP there are exist efficient combinatorial algorithms for this term (a list of examples is given in Table 1).

Name	Ops ($\bigoplus, \otimes$)	Backprop	Gradients
Log	LSE, $+$	Δ	$p(z_p = 1)$
Max	$\max, +$	Δ	$\arg\max_z$
K-Max	$k\max, +$	Δ	K-Argmax
Sample	LSE, $+$	$\sim$	$z \sim \mathrm{CRF}(\ell)$
K-Sample	LSE, $+$	$\sim$	K-Samples
Count	$\sum, \times$		
Entropy ($\mathbb{H}$)		See (Li and Eisner, 2009)	
Exp.		See (Li and Eisner, 2009)	
Sparsemax		See (Mensch and Blondel, 2018)	

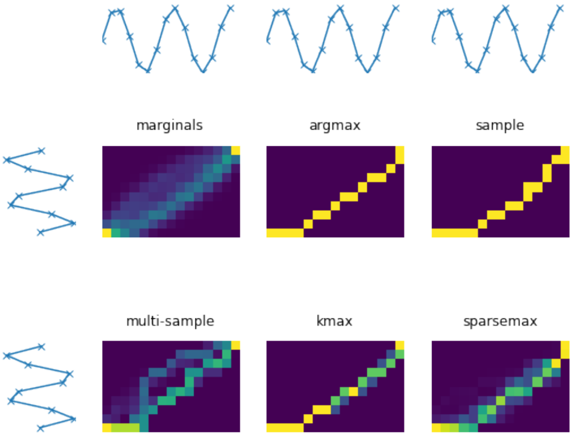


Table 2: (Top) Semirings implemented in Torch-Struct. *Backprop/Gradients* gives overridden backpropagation computation and value computed by this combination. (Bot) Example of gradients from different semirings on sequence alignment with dynamic time warping.

Derivatives of the log-partition again provide useful distributional properties. For instance, the marginal probabilities of parts are given by,

$$p(z_p = 1) = \frac{\exp \sum_{z: z_p = 1} z \cdot \ell}{\sum_{z' \in} \exp z' \cdot \ell} = \frac{\partial}{\partial \ell_p} A(\ell)$$

Similarly derivatives of A^* correspond to whether a part appears in the argmax structure, $\mathbb{I}(z_p^* = 1) = \frac{\partial}{\partial \ell_p} A^*(\ell)$.

While these gradient identities are well-known (Eisner, 2016), they are not commonly deployed in practice. Computing CRF properties is typically done through two-step specialized algorithms, such as forward-backward, inside-outside, or similar variants such as viterbi-backpointers (Jurafsky and Martin, 2014). Common wisdom is that these approaches are more efficient implementations.

However, we observe that recent engineering of faster gradient computation for deep learning has made gradient-based calculations competitive with hand-written calculations. In our experiments, we found that using these identities with autodiffer-

structures to ensure that properties hold. While this is intractable for large spaces, it can be done for small sets and was extremely useful for development.

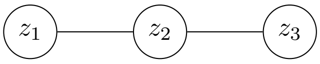
[2] This is a subgradient identity, but that deep learning libraries like PyTorch generally default to this value.

entiation was often faster, and much simpler, than custom two-pass approaches. Torch-Struct is thus designed around using gradients for distributional computations.

5.2 Dynamic Programming and Semirings

Torch-Struct is a collection of generic algorithms for CRF inference. Each CRF distribution object, CRF(ℓ), is constructed by providing $\ell \in \mathbb{R}^{|\mathcal{P}|}$ where the parts $\mathcal{P}$ are specific to the type of distribution. Internally, each distribution is implemented through a single function for computing the log-partition function $A(\ell)$. From this function, the library uses autodifferentiation and the identities from the previous section, to define a complete distribution object. The core models implemented by the library are shown in Table 1.

To make the approach concrete, we consider the example of the simplest structured model, a linear-chain CRF $p(z_1, z_2, z_3 \mid x)$.



The model has C labels per node with a length N utilizing a first-order linear-chain (Markov) model. This model has $N-1 \times C \times C$ parts corresponding to edges in the chain, and thus $\ell \in \mathbb{R}^{N-1 \times C \times C}$ log-potentials. The log-partition function $A(\ell)$ factors into two reduce computations,

$$\begin{aligned} A(\ell) &= \log \sum_{c_3,c_2} \exp \ell_{2,c_2,c_3} \sum_{c_1} \exp \ell_{1,c_1,c_2} \\ &= \mathrm{LSE}_{c_3,c_2}[\ell_{2,c_2,c_3} + [\mathrm{LSE}_{c_1} \ell_{1,c_1,c_2}]] \end{aligned}$$

Computing this function left-to-right using dynamic programming yields the standard *forward* algorithm for computing the log-partition of sequence models. As we have seen, the gradient with respect to ℓ produces marginals for each part, i.e. the probability of a specific labeled edge.

We can further extend the same function to support generic *semiring* dynamic programming (Goodman, 1999). A semiring is defined by a pair $(\oplus, \otimes)$ with commutative $\oplus$, distribution, and appropriate identities.

$$A(\ell) = \bigoplus_{c_3,c_2} [\ell_{2,c_2,c_3} \otimes [\bigoplus_{c_1} \ell_{1,c_1,c_2}]]$$

The log-partition utilizes $\oplus, \otimes = (\mathrm{LSE}, +)$, but we can substitute alternatives. For instance, utilizing the log-max semiring $(\max, +)$ in the forward algorithm yields the max score. As we have seen, its gradient with respect to ℓ is the argmax sequence, negating the need for a separate argmax (Viterbi) algorithm. Some distributional properties cannot be computed directly through gradient identities but still use a forward-backward style compute structure. For instance, sampling requires first computing the log-partition term and then sampling each part, (forward filtering / backward sampling). We can compute this value by overriding each backpropagation operation for the $\bigoplus$ to instead compute a sample.

Table 2 shows the set of semirings and backpropagation steps for computing different terms of interest. We note that many of the terms necessary in the case-study can be computed with variant semirings, negating the need for specialized algorithms.

6 Optimizations

Torch-Struct aims for computational and memory efficiency. Implemented naively, dynamic programming algorithms in Python are prohibitively slow. As such Torch-Struct provides key primitives to help batch and vectorize these algorithms to take advantage of GPU computation and to minimize the overhead of backpropagating through chart-based dynamic programmming. We discuss three optimizations: a) Parallel Scan, b) Vectorization, and c) Semiring Matrix Multiplications. Figure 3 shows the impact of these optimizations on the core algorithms.

Parallel Scan Inference The commutative properties of semiring algorithms allow flexibility in the order in which we compute $A(\ell)$. Typical implementations of dynamic programming algorithms are serial in the length of the sequence. On parallel hardware, an appealing approach is a parallel scan ordering (Särkkä and García-Fernández, 2019), typically used for computing prefix sums. To compute, $A(\ell)$ in this manner we first pad the sequence length N out to the nearest power of two, and then compute a balanced parallel tree over the parts, shown in Figure 4. Concretely each node layer would compute a semiring matrix multiplication, e.g. $\bigoplus_c \ell_{n,\cdot,c} \otimes \ell_{n+1,c,\cdot}$. Under this approach, assuming enough parallel cores, we only need $O(\log N)$ steps in Python and can use parallel operations for the rest. Similar parallel approach can also be used for computing sequence alignment and semi-Markov models.

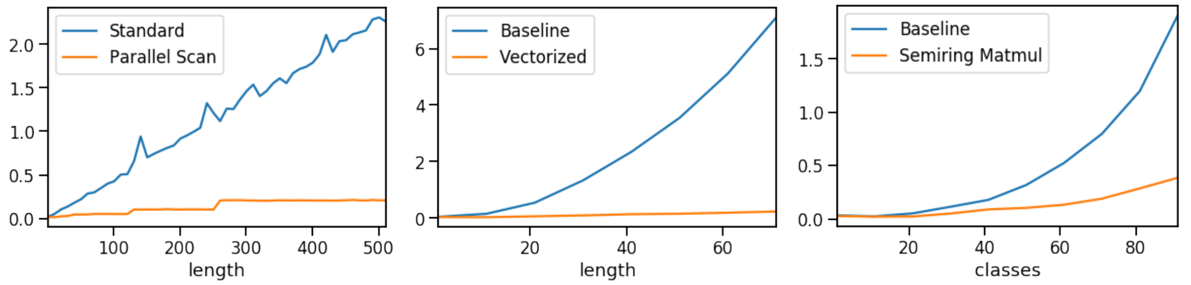


Figure 3: Speed impact of optimizations. Time is given in seconds for 10 runs with batch 16. (a) Speed of a linear-chain forward with 20 classes for lengths up to 500. Compares left-to-right ordering to parallel scan. (b) Speed of CKY inside with lengths up to 80. Compares inner loop versus vectorization. (c) Speed of linear-chain forward of length 20 with up to 100 classes. Compares broadcast-reduction versus CUDA semiring kernel. (Baseline memory is exhausted after 100 classes.)

Vectorization Computational complexity is even more of an issue for algorithms that cannot easily be parallelized. For example, parsing algorithms the generalize CKY are common in NLP. The CKY algorithm has a bottleneck that it must compute each width from 1 through N in serial; however internally each one of these steps can be vectorized. Assuming we have computed all inside spans of width less than d, computing the inside span of width d requires computing for all i,

$$C[i, i+d] \quad = \quad \bigoplus_{j=i}^{i+d-1} C[i, j] \otimes C[j+1, i+d]$$

In order to vectorize this loop over i, j, we need to reindex the chart. Instead of using a single chart C, we split it into two parts: one right-facing $C_r[i, d] = C[i, i+d]$ and one left facing, $C_l[i+d, N-d] = C[i, i+d]$. After this reindexing, the update can be written.

$$C_r[i, d] = \bigoplus_{j=1}^{j-1} C_r[i, j] \otimes C_l[i+d, N-d+j]$$

Unlike the original, this formula can easily be computed as a vectorized semiring dot product. This allows use to compute $C_r[\cdot, d]$ in one operation. Variants of this same approach can be used for many more complex dynamic programs.

Semiring Matrix Operations The two previous optimizations reduce most of the cost to semiring matrix multiplication. In the specific case of the $(\sum, \times)$ semiring these can be computed very efficiently using matrix multiplication, which is highly-tuned on GPU hardware. However, this semiring is not particularly useful and prone to underflow. For other semirings, such as log and max, these operations are either slow or very memory inefficient. For instance, for matrices T and U of sized $N \times M$ and $M \times O$, we can broadcast with $\otimes$ to a tensor of size $N \times M \times O$ and then reduce dim M by $\bigoplus$ at a huge memory cost.

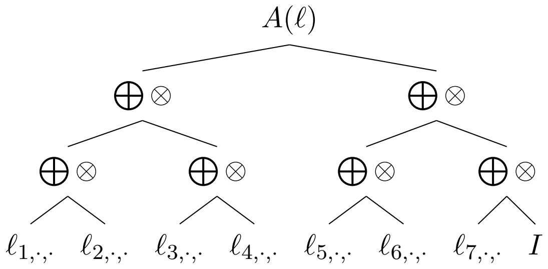


Figure 4: Parallel scan implementation of the linear-chain CRF inference algorithm (parallel forward). Here $\bigoplus \otimes$ represents a semiring matrix operation and I is padding to produce a balanced tree.

To avoid this issue, we implement custom CUDA kernels targeting fast and memory efficient tensor operations. For log, this corresponds to computing,

$$V_{m,o} = \log \sum_n \exp(T_{m,n} + U_{n,o} - q) + q$$

where $q = \max_n T_{m,n} + U_{n,o}$. To optimize this operation on GPU we utilize the TVM language (Chen et al., 2018) to layout the CUDA loops and tune it to hardware. This produces much faster operations, although still less efficient that matrix multiplication which is heavily customized to hardware.

7 Conclusion and Future Work

We present Torch-Struct, a library for deep structured prediction. The library achieves modularity through its adoption of a generic distributional

API, completeness by utilizing CRFs and semirings to make it easy to add new algorithms, and efficiency through core optimizations to vectorize important dynamic programming steps. In addition to the problems discussed so far, Torch-Struct also includes several other example implementations including supervised dependency parsing with BERT, unsupervised tagging, structured attention, and connectionist temporal classification (CTC) for speech. Code demonstrates that the model is able to replicate standard deep learning results, although we focus here on the fidelity and implementation approach of the core library. The full library is available at https://github.com/harvardnlp/pytorch-struct.

In the future, we hope to support research and production applications employing structured models. We also believe the library provides a strong foundation for building generic tools for interpretablity, control, and visualization through its probabilistic API. Finally, we hope to explore further optimizations to make core algorithms competitive with highly-optimized neural network components. These approaches provide a benchmark for improving autodifferentiation systems and extending their functionality to higher-order properties.

Acknowledgements

We thank Yoon Kim, Xiang Lisa Li, Sebastian Gehrmann, Yuntian Deng, and Justin Chiu for discussion and feedback on the project. The project was supported by NSF CAREER 1845664, NSF 1901030, and research awards by Sony and AWS.

References

Leonard E Baum and Ted Petrie. 1966. Statistical inference for probabilistic functions of finite state markov chains. *The annals of mathematical statistics*, 37(6):1554–1563.

Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research*, 20(1):973–978.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Tvm: end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537.

Joshua V Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A Saurous. 2017. Tensorflow distributions. *arXiv preprint arXiv:1711.10604*.

Greg Durrett and Dan Klein. 2015. Neural crf parsing. *arXiv preprint arXiv:1507.03641*.

Jason Eisner. 2000. Bilexical grammars and their cubic-time parsing algorithms. In *Advances in probabilistic and other parsing technologies*, pages 29–61. Springer.

Jason Eisner. 2016. Inside-outside and forward-backward algorithms are just backprop (tutorial paper). In *Proceedings of the Workshop on Structured Prediction for NLP*, pages 1–17.

Jason Eisner, Eric Goldlust, and Noah A Smith. 2004. Dyna: A declarative language for implementing dynamic programs. In *Proceedings of the ACL 2004 on Interactive poster and demonstration sessions*, pages 32–es.

Jenny Rose Finkel, Alex Kleeman, and Christopher D Manning. 2008. Efficient, feature-based, conditional random field parsing. In *Proceedings of ACL-08: HLT*, pages 959–967.

Zoubin Ghahramani and Michael I Jordan. 1996. Factorial hidden markov models. In *Advances in Neural Information Processing Systems*, pages 472–478.

Sharon Goldwater and Tom Griffiths. 2007. A fully bayesian approach to unsupervised part-of-speech tagging. In *Proceedings of the 45th annual meeting of the association of computational linguistics*, pages 744–751.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4):573–605.

Serhii Havrylov, Germán Kruszewski, and Armand Joulin. 2019. Cooperative learning of disjoint syntax and semantics. *arXiv preprint arXiv:1902.09393*.

Thorsten Joachims. 2008. Svmstruct: Support vector machine for complex outputs.

Matthew J Johnson, David K Duvenaud, Alex Wiltschko, Ryan P Adams, and Sandeep R Datta. 2016. Composing graphical models with neural networks for structured representations and fast inference. In *Advances in neural information processing systems*, pages 2946–2954.

Dan Jurafsky and James H Martin. 2014. Speech and language processing. vol. 3.

Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*.

Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. 2017. Structured attention networks. *CoRR*, abs/1702.00887.

Terry Koo, Amir Globerson, Xavier Carreras Pérez, and Michael Collins. 2007. Structured prediction models via the matrix-tree theorem. In *Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, pages 141–150.

Taku Kudo. 2005. Crf++: Yet another crf toolkit. *http://crfpp. sourceforge. net/.*

John Lafferty, Andrew McCallum, and Fernando CN Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data.

Zhifei Li and Jason Eisner. 2009. First-and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 1-Volume 1*, pages 40–51. Association for Computational Linguistics.

André FT Martins, Noah A Smith, Eric P Xing, Pedro MQ Aguiar, and Mário AT Figueiredo. 2010. Turbo parsers: Dependency parsing by approximate variational inference. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 34–44. Association for Computational Linguistics.

Ryan McDonald, Fernando Pereira, Kiril Ribarov, and Jan Hajič. 2005. Non-projective dependency parsing using spanning tree algorithms. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, pages 523–530. Association for Computational Linguistics.

Arthur Mensch and Mathieu Blondel. 2018. Differentiable dynamic programming for structured prediction and attention. *arXiv preprint arXiv:1802.03676*.

Andreas C Müller and Sven Behnke. 2014. Pystruct: learning structured prediction in python. *The Journal of Machine Learning Research*, 15(1):2055–2060.

Nikita Nangia and Samuel R Bowman. 2018. Listops: A diagnostic dataset for latent tree learning. *arXiv preprint arXiv:1804.06028*.

Saul B Needleman and Christian D Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453.

Naoaki Okazaki. 2007. Crfsuite: a fast implementation of conditional random fields (crfs).

Steven J Rennie, Etienne Marcheret, Youssef Mroueh, Jerret Ross, and Vaibhava Goel. 2017. Self-critical sequence training for image captioning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7008–7024.

Sunita Sarawagi and William W Cohen. 2005. Semi-markov conditional random fields for information extraction. In *Advances in neural information processing systems*, pages 1185–1192.

Simo Särkkä and Ángel F García-Fernández. 2019. Temporal parallelization of bayesian filters and smoothers. *arXiv preprint arXiv:1905.13002*.

Christoph Tillmann and Hermann Ney. 2003. Word reordering and a dynamic programming beam search algorithm for statistical machine translation. *Computational linguistics*, 29(1):97–133.

Stephan Vogel, Hermann Ney, and Christoph Tillmann. 1996. Hmm-based word alignment in statistical translation. In *Proceedings of the 16th conference on Computational linguistics-Volume 2*, pages 836–841. Association for Computational Linguistics.

Sam Wiseman, Stuart M Shieber, and Alexander M Rush. 2018. Learning neural templates for text generation. *arXiv preprint arXiv:1808.10122*.

Jie Yang and Yue Zhang. 2018. Ncrf++: An open-source neural sequence labeling toolkit. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*.

Dani Yogatama, Phil Blunsom, Chris Dyer, Edward Grefenstette, and Wang Ling. 2016. Learning to compose words into sentences with reinforcement learning. *arXiv preprint arXiv:1611.09100*.