SynJax: Structured Probability Distributions for JAX

Miloš Stanojević
Google DeepMind
stanojevic@google.com
Laurent Sartran
Google DeepMind
lsartran@google.com

Abstract
The development of deep learning software libraries enabled significant progress in the field by allowing users to focus on modeling, while letting the library take care of the tedious and time-consuming task of optimizing execution for modern hardware accelerators. However, this has benefited only particular types of deep learning models, such as Transformers, whose primitives map easily to the vectorized computation. The models that explicitly account for structured objects, such as trees and segmentations, did not benefit equally because they require custom algorithms that are difficult to implement in a vectorized form.

SynJax directly addresses this problem by providing an efficient vectorized implementation of inference algorithms for structured distributions covering alignment, tagging, segmentation, constituency trees and spanning trees. This is done by exploiting the connection between algorithms for automatic differentiation and probabilistic inference. With SynJax we can build large-scale differentiable models that explicitly model structure in the data. The code is available at https://github.com/google-deepmind/synjax.

1 Introduction
In many domains, data can be seen as having some structure explaining how its parts fit into a larger whole. This structure is often latent, and it varies depending on the task. For examples of discrete structures in natural language consider Figure 1. The words together form a sequence. Each word in a sequence is assigned a part-of-speech tag. These tags are dependent on each other, forming a linear chain marked in red. The words in the sentence can be grouped together into small disjoint contiguous groups by sentence segmentation, shown with bubbles. A deeper analysis of language would show that the groupings can be done recursively and thereby produce a syntactic tree structure. Structures can also relate two languages. For instance, in the same figure, a Japanese translation can be mapped to an English source by an alignment.

These structures are not specific to language. Similar structures appear in biology as well. Nucleotides of any two RNA sequences are matched with monotone alignment (Needleman and Wunsch, 1970; Wang and Xu, 2011), genomic data is segmented into contiguous groups (Day et al., 2007) and tree-based models of RNA capture the hierarchical nature of the protein folding process (Sakakibara et al., 1994; Hockenmaier et al., 2007; Huang et al., 2019).

Most contemporary deep learning models attempt to predict output variables directly from the input without any explicit modeling of the intermediate structure. Modeling structure explicitly could improve these models in multiple ways. First, it could allow for better generalization through the right inductive biases (Dyer et al., 2016; Sartran et al., 2022). This would improve not only sample efficiency but also downstream performance (Bastings et al., 2017; Nadejde et al., 2017; Bisk and Tran, 2018). Explicit modeling of structure can also enable incorporation of problem specific algorithms (e.g. finding shortest paths; Pogančić et al., 2020; Niepert et al., 2021) or constraints (e.g. enforcing alignment Mena et al., 2018 or enforcing compositional calculation Havrylov et al., 2019). Discrete structure also allows for better interpretability of the model’s decisions (Bastings et al., 2017; Nadejde et al., 2017; Bisk and Tran, 2018). Explicit modeling of structure could also improve these models in multiple ways. First, it could allow for better generalization through the right inductive biases (Dyer et al., 2016; Sartran et al., 2022). This would improve not only sample efficiency but also downstream performance (Bastings et al., 2017; Nadejde et al., 2017; Bisk and Tran, 2018). Explicit modeling of structure can also enable incorporation of problem specific algorithms (e.g. finding shortest paths; Pogančić et al., 2020; Niepert et al., 2021) or constraints (e.g. enforcing alignment Mena et al., 2018 or enforcing compositional calculation Havrylov et al., 2019). Discrete structure also allows for better interpretability of the model’s decisions (Bastings et al., 2017; Nadejde et al., 2017; Bisk and Tran, 2018).
et al., 2019). Finally, sometimes structure is the end goal of learning itself – for example we may know that there is a hidden structure of a particular form explaining the data, but its specifics are not known and need to be discovered (Kim et al., 2019; Paulus et al., 2020).

Auto-regressive models are the main approach used for modeling sequences. Non-sequential structures are sometimes linearized and approximated with a sequential structure (Choe and Charniak, 2016). These models are powerful as they do not make any independence assumptions and can be trained on large amounts of data. While sampling from auto-regressive models is typically tractable, other common inference problems like finding the optimal structure or marginalizing over hidden variables are not tractable. Approximately solving these tasks with auto-regressive models requires using biased or high-variance approximations that are often computationally expensive, making them difficult to deploy in large-scale models.

Alternative to auto-regressive models are models over factor graphs that factorize in the same way as the target structure. These models can efficiently compute all inference problems of interest exactly by using specialized algorithms. Despite the fact that each structure needs a different algorithm, we do not need a specialized algorithm for each inference task (argmax, sampling, marginals, entropy etc.). As we will show later, SynJax uses automatic differentiation to derive many quantities from just a single function per structure type.

Large-scale deep learning has been enabled by easy to use libraries that run on hardware accelerators. Research into structured distributions for deep learning has been held back by the lack of ergonomic libraries that would provide accelerator-friendly implementations of structure components – especially since these components depend on algorithms that often do not map directly onto available deep learning primitives, unlike Transformer models. This is the problem that SynJax addresses by providing easy to use structure primitives that compose within JAX machine learning framework.

To see how easy it is to use SynJax consider example in Figure 2. This code implements a policy gradient loss that requires computing multiple quantities – sampling, argmax, entropy, log-probability – each requiring a different algorithm. In this concrete code snippet, the structure is a non-projective directed spanning tree with a single root edge constraint. Because of that SynJax will:

- compute argmax with Tarjan’s (1977) maximum spanning tree algorithm adapted for single root edge trees (Stanojević and Cohen, 2021),
- sample with Wilson’s (1996) sampling algorithm for single root trees (Stanojević, 2022),
- compute entropy with Matrix-Tree Theorem (Tutte, 1984) adapted for single root edge trees (Koo et al., 2007; Zmigrod et al., 2021).

If the user wants only to change slightly the tree requirements to follow the projectivity constraint they only need to change one flag and SynJax will in the background use completely different algorithms that are appropriate for that structure: it will use Kuhlmann’s algorithm (2011) for argmax and variations of Eisner’s (1996) algorithm for other quantities. The user does not need to implement any of those algorithms or even be aware of their specifics, and can focus on the modeling side of the problem.

2 Structured Distributions

Distributions over most structures can be expressed with factor graphs – bipartite graphs that have random variables and factors between them. We associate to each factor a non-negative scalar, called potential, for each possible assignment of the random variables that are in its neighbourhood. The potential of the structure is a product of its factors:

$$\phi(t) = \prod_{e \in t} \phi(e)$$

where $t$ is a structure, $e$ is a factor/part, and $\phi(\cdot)$ is the potential function. The probability of a struc-
ture can be found by normalizing its potential:

\[ p(t) = \frac{\prod_{e \in t} \phi(e)}{\sum_{t' \in T} \prod_{e' \in t'} \phi(e')} = \frac{\phi(t)}{Z} \]  

(2)

where \( T \) is the set of all possible structures and \( Z \) is a normalization often called partition function. This equation can be thought of as a softmax equivalent over an extremely large set of structured outputs that share sub-structures (Sutton and McCallum, 2007; Mihaylova et al., 2020).

3 Computing Probability of a Structure and Partition Function

Equation 2 shows the definition of the probability of a structure in a factor graph. Computing the numerator is often trivial. However, computing the denominator, the partition function, is the complicated and computationally demanding part because the set of valid structures \( T \) is usually exponentially large and require specialized algorithms for each type of structure. As we will see later, the algorithm for implementing the partition function accounts for the majority of the code needed to add support for a structured distribution, as most of the other properties can be derived from it. Here we document the algorithms for each structure.

3.1 Sequence Tagging

Sequence tagging can be modelled with Linear-Chain CRF (Lafferty et al., 2001). The partition function for linear-chain models is computed with the forward algorithm (Rabiner, 1990). The computational complexity is \( \mathcal{O}(m^2n) \) for \( m \) tags and sequence of length \( n \). Särkkä and García-Fernández (2021) have proposed a parallel version of this algorithm that has parallel computational complexity \( \mathcal{O}(mn^3 \log n) \) which is efficient for \( m \ll n \). Rush (2020) reports a speedup using this parallel method for Torch-Struct, however in our case the original forward algorithm gave better performance both in terms of speed and memory.

The SynJax implementation of Linear-Chain CRF supports having a different transition matrix for each time step which gives greater flexibility needed for implementing models like LSTM-CNN-CRF (Ma and Hovy, 2016) and Neural Hidden Markov Model (Tran et al., 2016).

3.2 Segmentation with Semi-Markov CRF

Joint segmentation and tagging can be done with a generalization of linear-chain called Semi-Markov CRF (Sarawagi and Cohen, 2004; Abdel-Hamid et al., 2013; Lu et al., 2016). It has a similar parametrization with transition matrices except that here transitions can jump over multiple tokens. The partition function is computed with an adjusted version of the forward algorithm that runs in \( \mathcal{O}(sm^2n) \) where \( s \) is the maximal size of a segment.

3.3 Alignment Distributions

Alignment distributions are used in time series analysis (Cuturi and Blondel, 2017), RNA sequence alignment (Wang and Xu, 2011), semantic parsing (Lyu and Titov, 2018) and many other areas.

3.3.1 Monotone Alignment

Monotone alignment between two sequences of lengths \( n \) and \( m \) allows for a tractable partition function that can be computed in \( \mathcal{O}(nm) \) time using the Needleman-Wunsch (1970) algorithm.

3.3.2 CTC

Connectionist Temporal Classification (CTC, Graves et al., 2006; Hannun, 2017) is a monotone alignment model widely used for speech recognition and non-auto-regressive machine translation models. It is distinct from the standard monotone alignment because it requires special treatment of the blank symbol that provides jumps in the alignment table. It is implemented with an adjusted version of Needleman-Wunsch algorithm.

3.3.3 Non-Monotone 1-on-1 Alignment

This is a bijective alignment that directly maps elements between two sets given their matching score. Computing partition function for this distribution is intractable (Valiant, 1979), but we can compute some other useful quantities (see Section 5).

3.4 Constituency Trees

3.4.1 Tree-CRF

Today’s most popular constituency parser by Kitaev et al. (2019) uses a global model with factors defined over labelled spans. Stern et al. (2017) have shown that inference in this model can be done efficiently with a custom version of the CKY algorithm in \( \mathcal{O}(mn^2 + n^3) \) where \( m \) is number of non-terminals and \( n \) is the sentence length.

3.4.2 PCFG

Probabilistic Context-Free Grammars (PCFG) are a generative model over constituency trees where each grammar rule is associated with a locally normalized probability. These rules serve as a template
which, when it gets expanded, generates jointly a constituency tree together with words as leaves.

SynJax computes the partition function using a vectorized form of the CKY algorithm that runs in cubic time. Computing a probability of a tree is in principle simple: just enumerate the rules of the tree, look up their probability in the grammar and multiply the found probabilities. However, extracting rules from the set of labelled spans requires many sparse operations that are non-trivial to vectorize. We use an alternative approach where we use sticky span log-potentials to serve as a mask for each constituent: constituents that are part of the tree have sticky log-potentials 0 while those that are not are $-\infty$. With sticky log-potentials set in this way computing log-partition provides a log-probability of a tree of interest.

### 3.4.3 TD-PCFG

Tensor-Decomposition PCFG (TD-PCFG, Cohen et al., 2013; Yang et al., 2022) uses a lower rank tensor approximation of PCFG that makes inference with much larger number of non-terminals feasible.

### 3.5 Spanning Trees

Spanning trees appear in the literature in many different forms and definitions. We take a spanning tree to be any subgraph that connects all nodes and does not have cycles. We divide spanning tree CRF distributions by the following three properties:

- **Directed or Undirected**: Undirected spanning trees are defined over symmetric weighted adjacency matrices i.e. over undirected graphs. Directed spanning trees are defined over directed graphs with special root node.

- **Projective or Non-Projective**: Projectivity is a constraint that appears often in NLP. It constrains the spanning tree over words not to have crossing edges. Non-projective spanning tree is just a regular spanning tree – i.e. it may not satisfy the projectivity constraint.

- **Single Root Edge or Multi Root Edges**: NLP applications usually require that there can be only one edge coming out of the root (Zmigrod et al., 2020). Single root edge spanning trees satisfy that constraint.

Each of these choices has direct consequences on which algorithm should be used for probabilistic inference. SynJax abstracts away this from the user and offers a unified interface where the user only needs to provide the weighted adjacency matrix and set the three mentioned boolean values. Given the three booleans SynJax can pick the correct and most optimal algorithm. In total, these parameters define distributions over 8 different types of spanning tree structures all unified in the same interface. We are not aware of any other library providing this set of unified features for spanning trees.

We reduce undirected case to the rooted directed case due to bijection. For projective rooted directed spanning trees we use Eisner’s algorithm for computation of the partition function (Eisner, 1996). The partition function of Non-Projective spanning trees is computed using Matrix-Tree Theorem (Tutte, 1984; Koo et al., 2007; Smith and Smith, 2007).

### 4 Computing Marginals

In many cases we would like to know the probability of a particular part of structure appearing, regardless of the structure that contains it. In other words, we want to marginalize (i.e. sum) the probability of all the structures that contain that part:

$$p(e) = \sum_{t \in T} 1[e \in t] \ p(t) = \sum_{t' \in T_e} p(t')$$

(3)

where $1[\cdot]$ is the indicator function, $T$ is the set of all structures and $T_e$ is the set of structures that contain factor/part $e$.

Computing these factors was usually done using specialized algorithms such as Inside-Outside or Forward-Backward. However, those solutions do not work on distributions that cannot use belief propagation like Non-Projective Spanning Trees. A more general solution is to use an identity that relates gradients of factor’s potentials with respect to the log-partition function:

$$p(e) = \frac{\partial \log Z}{\partial \phi(e)}$$

(4)

This means that we can use any differentiable implementation of log-partition function as a forward pass and apply backpropagation to compute the marginal probability (Darwiche, 2003). Eisner (2016) has made an explicit connection that “Inside-Outside and Forward-Backward algorithms are just backprop”. This approach also works for Non-Projective Spanning Trees that do not fit belief propagation framework (Zmigrod et al., 2021).

For template models like PCFG, we use again the sticky log-potentials because usually we are
not interested in marginal probability of the rules but in the marginal probability of the instantiated constituents. The derivative of log-partition with respect to the constituent’s sticky log-potential will give us marginal probability of that constituent.

5 Computing Most Probable Structure

For finding the score of the highest scoring structure we can just run the same belief propagation algorithm for log-partition, but with the max-plus semiring instead of the log-plus semiring (Goodman, 1999). To get the most probable structure, and not just its potential, we can compute the gradient of part potentials with respect to the viterbi structure potential (Rush, 2020).

The only exceptions to this process are non-monotone alignments and spanning trees because they do fit easily in belief propagation framework. For the highest scoring non-monotone alignment, we use the Jonker–Volgenant algorithm as implemented in SciPy (Crouse, 2016; Virtanen et al., 2020). Maximal projective spanning tree can be found by combining Eisner’s algorithm with max-plus semiring, but we have found Kuhlmann’s tabulated arc-hybrid algorithm to be much faster (Kuhlmann et al., 2011) (see Figure 4 in the appendix). This algorithm cannot be used for any inference task other than argmax because it allows for spurious derivations. To enforce single-root constraint with Kuhlmann’s algorithm we use the Reweighting trick from Stanojević and Cohen (2021). For non-projective spanning trees SynJax uses a combination of Reweighting+Tarjan algorithm as proposed in Stanojević and Cohen (2021).

6 Sampling a Structure

Strictly speaking, there is no proper sampling semiring because semirings cannot have non-deterministic output. However, we can still use the semiring framework and make some aspect of them non-deterministic. Aziz (2015) and Rush (2020) use a semiring that in the forward pass behaves like a log-semiring, but in the backward pass instead of computing the gradient it does sampling. This is in line of how forward-filtering backward-sampling algorithm works (Murphy, 2012, §17.4.5).

Non-Projective Spanning Trees do not support the semiring framework so we use custom algorithms for them described in Stanojević (2022). It contains Colbourn’s algorithm that has a fixed runtime of \(O(n^3)\) but is prone to numerical issues because it requires matrix-inversion (Colbourn et al., 1996), and Wilson’s algorithm that is more numerically stable but has a runtime that depends on concrete values of log-potentials (Wilson, 1996). SynJax also supports vectorized sampling without replacement (SWOR) from Stanojević (2022).

7 Differentiable Sampling

The mentioned sampling algorithms provide unbiased samples of structures useful for many inference tasks, but they are not differentiable because the gradient of sampling from discrete distributions is zero almost everywhere. This problem can be addressed with log-derivative trick from REINFORCE algorithm (Williams, 1992), but that provides high variance estimates of gradients. To address this problem there have been different proposals for differentiable sampling algorithms that are biased but can provide low-variance estimates of gradients. SynJax implements majority of the main approaches in the literature including structured attention (Kim et al., 2017), relaxed dynamic programming (Mensch and Blondel, 2018), Perturb-and-MAP (Corro and Titov, 2019), Gumbel-CRF (Fu et al., 2020), Stochastic Softmax-Tricks (Paulus et al., 2020), and Implicit Maximum-Likelihood estimation (Niepert et al., 2021). It also include different noise distributions for perturbations models, including Sum-of-Gamma noise (Niepert et al., 2021) that is particularly suited for structured distributions.

8 Entropy and KL Divergence

To compute the cross-entropy and KL divergence, we will assume that the two distributions factorize in exactly the same way. Like some other properties, cross-entropy can also be computed with the appropriate semirings (Hwa, 2000; Eisner, 2002; Cortes et al., 2008; Chang et al., 2023), but those approaches would not work on Non-Projective Spanning Tree distributions. There is a surprisingly simple solution that works across all distributions that factorize in the same way and has appeared in a couple of works in the past (Li and Eisner, 2009; Martins et al., 2010; Zmigrod et al., 2021). Here
we give a full derivation for cross-entropy:

\[
H(p, q) = - \sum_{t \in T} p(t) \log q(t)
\]

\[
= \log Z_q - \sum_{t \in T} p(t) \sum_{e \in t} \log \phi_q(e)
\]

\[
= \log Z_q - \sum_{e \in E} p(e) \log \phi_q(e)
\]

This reduces the computation of cross-entropy to finding marginal probabilities of one distribution, and finding log-partition of the other – both of which can be computed efficiently for all distributions in SynJax. Given the method for computing cross-entropy, finding entropy is trivial:

\[
H(p) = H(p, p)
\]

KL divergence is easy to compute too:

\[
D_{KL}(p || q) = H(p, q) - H(p)
\]

9 Library Design

Each distribution has different complex shape constraints which makes it complicated to document and implement all the checks that verify that the user has provided the right arguments. The jaxtyping library\(^1\) was very valuable in making SynJax code concise, documented and automatically checked.

Structured algorithms require complex broadcasting, reshaping operations and application of semirings. To make this code simple, we took the einsum implementation from the core JAX code and modified it to support arbitrary semirings. This made the code shorter and easier to read.

Most inference algorithms apply a large number of elementwise and reshaping operations that are in general fast but create a large number of intermediate tensors that occupy memory. To speed this up we use checkpointing (Griewank, 1992) to avoid memorization of tensors that can be recomputed quickly. That has improved memory usage and speed, especially on TPUs.

All functions that could be vectorized are written in pure JAX. Those that cannot, like Wilson sampling (1996) and Tarjan’s algorithm (1977), are implemented with Numba (Lam et al., 2015).

---

\[^1\]https://github.com/google/jaxtyping

All SynJax distributions inherit from Equinox modules (Kidger and Garcia, 2021) which makes them simultaneously PyTrees and dataclasses. Thereby all SynJax distributions can be transformed with jax.vmap and are compatible with any JAX neural framework, e.g. Haiku and Flax.

10 Comparison to alternative libraries

JAX has a couple of libraries for probabilistic modeling. Distrax (Babuschkin et al., 2020) and Tensorflow-Probability JAX substrate (Dillon et al., 2017) provide continuous distributions. NumPyro (Phan et al., 2019) and Oryx provide probabilistic programming. DynaMax (Chang et al., 2022) brings state space models to JAX and includes an implementation of HMMs.

PGMax (Zhou et al., 2023) is a JAX library that supports inference over arbitrary factor graphs by using loopy belief propagation. After the user builds the desired factor graph, PGMax can do automatic inference over it. For many structured distributions building a factor graph is the difficult part of implementation because it may require a custom algorithm (e.g. CKY or Needleman–Wunsch). SynJax implements those custom algorithms for each of the supported structures. With SynJax the user only needs to provide the parameters of the distribution and SynJax will handle both building of the factor graph and inference over it. For all the included distributions, SynJax also provides some features not covered by PGMax, such as unbiased sampling, computation of entropy, cross-entropy and KL divergence.

Optax (Babuschkin et al., 2020) provides CTC loss implementation for JAX but without support for computation of optimal alignment, marginals over alignment links, sampling alignments etc.

All the mentioned JAX libraries focus on continuous or categorical distributions and, with the exception of HMMs and CTC loss, do not contain distributions provided by SynJax. SynJax fills

---

Table 1: Comparison against Torch-Struct with respect to lines of code for log-partition and relative speedup in the computation of marginal probabilities.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Torch-Struct</th>
<th>SynJax</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear-Chain-CRF</td>
<td>32</td>
<td>15</td>
<td>(46%)</td>
</tr>
<tr>
<td>Semi-Markov CRF</td>
<td>54</td>
<td>15</td>
<td>(27%)</td>
</tr>
<tr>
<td>Tree-CRF</td>
<td>21</td>
<td>14</td>
<td>(66%)</td>
</tr>
<tr>
<td>PCFG</td>
<td>51</td>
<td>36</td>
<td>(76%)</td>
</tr>
<tr>
<td>Projective CRF</td>
<td>70</td>
<td>54</td>
<td>(77%)</td>
</tr>
<tr>
<td>Non-Projective CRF</td>
<td>60</td>
<td>8</td>
<td>(16%)</td>
</tr>
</tbody>
</table>
this gap in the JAX ecosystem and enables easier construction of structured probability models.

The most comparable library in terms of features is Torch-Struct (Rush, 2020) that targets PyTorch as its underlying framework. Torch-Struct, just like SynJax, uses automatic differentiation for efficient inference. We will point out here only the main differences that would be of relevance to users. SynJax supports larger number of distributions and inference algorithms and provides a unified interface to all of them. It also provides reproducible sampling through controlled randomness seeds. SynJax has a more general approach to computation of entropy that does not depend on semirings and therefore applies to all distributions. SynJax is fully implemented in Python and compiled with jax.jit and numba.jit while Torch-Struct does not use any compiler optimizations except a custom CUDA kernel for semiring matrix multiplication. If we compare lines of code and speed (Table 1) we can see that SynJax is much more concise and faster than Torch-Struct (see Appendix A for details).

SynJax also provides the fastest and most feature rich implementation of spanning tree algorithms. So far the most competitive libraries for spanning trees were by Zmigrod et al. and Stanojević and Cohen. SynJax builds on Stanojević and Cohen code and annotates it with Numba instructions which makes it many times faster than any other alternative (see Figure 3 in the appendix).

11 Conclusion

One of the main challenges in creating deep neural models over structured distributions is the difficulty of their implementation on modern hardware accelerators. SynJax addresses this problem and makes large scale training of structured models feasible and easy in JAX. We hope that this will encourage research into finding alternatives to auto-regressive modeling of structured data.

Limitations

SynJax is quite fast, but there are still some areas where the improvements could be made. One of the main speed and memory bottlenecks is usage of big temporary tensors in the dynamic programming algorithms needed for computation of log-partition function. This could be optimized with custom kernels written in Pallas.\(^2\) There are some speed gains that would conceptually be simple but they depend on having a specialized hardware. For instance, matrix multiplication with semirings currently does not use hardware acceleration for matrix multiplication, such as TensorCore on GPU, but instead does calculation with regular CUDA cores. We have tried to address this with log-einsum-exp trick (Peharz et al., 2020) but the resulting computation was less numerically precise than using a regular log-semiring with broadcasting. Maximum spanning tree algorithm would be much faster if it could be vectorized – currently it’s executing as an optimized Numba CPU code.

Acknowledgements

We are grateful to Chris Dyer, Aida Nematzadeh and other members of language team in Google DeepMind for early comments on the draft of this work. We appreciate Patrick Kidger’s work on Equinox and Jaxtyping that made development of SynJax much easier. We also appreciate that Sasha Rush open-sourced Torch-Struct, a library that influenced many aspects of SynJax.

References


Distribution | parameters
--- | ---
CTC | $b=64$, $n=124$, $l=512$
Alignment CRF | $b=16$, $n=256$, $m=256$
Semi-Markov CRF | $b=1$, $n=64$, $nt=32$, $k=8$
Tree CRF | $b=128$, $n=128$, $nt=128$
Linear-Chain CRF | $b=128$, $n=256$, $nt=32$
PCFG | $b=1$, $n=48$, $nt=64$, $pt=96$
HMM | $b=1$, $n=128$, $nt=32$
Non-Projective CRF | $b=1024$, $n=128$
Projective CRF | $b=128$, $n=128$

Table 2: Sizes of tested distributions.

### A Empirical comparisons

#### A.1 Comparison with Torch-Struct

We compared against the most recent Torch-Struct commit from 30 Jan 2022. To make Torch-Struct run faster we have also installed its specialized kernel for semiring matrix multiplication *genbmm* from its most recent commit from 11 Oct 2021. While Torch-Struct supports some of the same distributions as SynJax we did not manage to do speed comparison over all of them. For example, AlignmentCRF of Torch-Struct was crashing due to mismatch of PyTorch, Torch-Struct and *genbmm* changes about in-place updates. We compile SynJax with jax.jit and during benchmarking do not count the time that is taken for compilation because it needs to be done only once. We also tried to compile Torch-Struct using TorchScript by tracing but that did not work out of the box. Comparisons are done on A100 GPU on Colab Pro+. The results are shown in Table 1 in the main text. Table 2 shows sizes of the distributions being tested.

#### A.2 Comparison with Zmigrod et al.

Non-Projective spanning trees present a particular challenge because they cannot be vectorized easily due to dynamic structures that are involved in the algorithm. The main algorithms and libraries for parsing this type of trees are from Zmigrod et al. (2020)\(^3\) and Stanojević and Cohen (2021)\(^6\). The first one is expressed as a recursive algorithm, while the second one operates over arrays of fixed size in iterative way. This makes Stanojević and Cohen algorithm much more amendable to Numba optimization. We took that code and just annotated

\(^3\)[https://github.com/harvardnlp/pytorch-struct](https://github.com/harvardnlp/pytorch-struct)

\(^4\)[https://github.com/harvardnlp/genbmm](https://github.com/harvardnlp/genbmm)

\(^5\)[https://github.com/rycolab/spanningtrees](https://github.com/rycolab/spanningtrees)

it with Numba primitives. This made the algorithm significantly faster, especially for big graphs, as can be seen from Figure 3.

![Figure 3: Speed comparison of Non-Projective Spanning Tree libraries.](image)

**A.3 Comparison of Maximum Projective Spanning Tree Algorithms**

Eisner’s algorithm is virtually the only projective parsing algorithm actively used, if we do not count the transition based parsers. We have found that replacing Eisner’s algorithm with Kuhlmann et al. (2011) tabulation of arc-hybrid algorithm can provide large speed gains both on CPU and GPU. See Figure 4. In this implementation graph size does not make a big difference because it is implemented in a vectorized way so most operations are parallelized.

![Figure 4: Speed comparison of Projective Maximum Spanning Tree algorithms.](image)