Journal of Machine Learning Research 23 (2022) 1-5 Submitted 2/21; Revised 6/22; Published 9/22

JsonGrinder.jl: automated differentiable neural architecture for embedding arbitrary JSON data

Šimon Mandlík SIMON.MANDLIK@GMAIL.COM
AIC, FEE, Czech Technical University in Prague
Avast Software s.r.o.

Matěj Račinský RACINSKY.MATEJ@SEZNAM.CZ
Avast Software s.r.o.

Viliam Lisý VILIAM.LISY@AVAST.COM
AIC, FEE, Czech Technical University in Prague
Avast Software s.r.o.

Tomáš Pevný PEVNAK@PROTONMAIL.CH
AIC, FEE, Czech Technical University in Prague
Avast Software s.r.o.

Editor: Andreas Mueller

Abstract

Standard machine learning (ML) problems are formulated on data converted into a suitable tensor representation. However, there are data sources, for example in cybersecurity, that are naturally represented in a unifying hierarchical structure, such as XML, JSON, and Protocol Buffers. Converting this data to a tensor representation is usually done by manual feature engineering, which is laborious, lossy, and prone to bias originating from the human inability to correctly judge the importance of particular features. `JsonGrinder.jl` is a library automating various ML tasks on these difficult sources. Starting with an arbitrary set of JSON samples, it automatically creates a differentiable ML model (called HMILnet), which embeds raw JSON samples into a fixed-size tensor representation. This embedding network can be naturally extended by an arbitrary ML model expecting tensor inputs in order to perform classification, regression, or clustering.

1. Motivation

```
{ "mac": "00:04:4b:a9:c1:f3",
  "ip": "192.168.1.122",
  "services": [{ "protocol": "udp", "port": 5353 },
               { "protocol": "tcp", "port": 6466 }],
  "upnp": [{ "model_name": "AirReceiver",
             "manufacturer": "SoftMedia Inc.",
             "model_description": "AirReceiver - Media Renderer",
             "services": ["urn:upnp-org:serviceId:AVTransport",
                          "urn:upnp-org:serviceId:RenderingControl"]},
           { "model_name": "SHIELD Android TV",
             "manufacturer": "NVIDIA",
             "services": []}],
  "mdns_services": ["_airplay._tcp.local.",
                    "_nv_shield_remote._tcp.local."]}
```

Fig. 1: A part of JSON sample from the Device ID challenge (CSP, 2019).

The last decade has witnessed a departure from feature engineering to end-to-end systems taking raw data as an input. It substantially reduced the human effort and increased performance for example in image recognition (Krizhevsky et al., 2017), natural language processing (Devlin et al., 2019), or game-playing tasks (Silver et al., 2017). There is a plethora of algorithms (and libraries) for creating classifiers, regressors, and other models when thr raw input is a fixed-dimensional tensor (images), sequences (text) or general graphs. In contrast, a lot of data used in the enterprise sector (e.g., data

exchanged by web services) are stored in a hierarchically structured serialization formats like JSON, XML, Protocol Buffer (Varda, 2008), or Message Pack (Furuhashi, 2010). Its structure resembles a tree with leaves being strings, numbers, or other primitive types; and internal nodes forming either **arbitrarily long** lists of subtrees (e.g. `services` in Fig. 1) or **possibly incomplete** sets of key-value pairs (e.g., elements of `upnp` in Fig. 1). Let us call them *Hierarchical Multiple Instance Learning (HMIL)* data, which refers to the hierarchical structure and to multiple instance learning problems, as introduced in Dietterich et al. (1997). HMIL data cannot be naturally represented as fixed vectors without the laborious and lossy feature engineering, and it cannot be represented as plain sequences without losing the key information captured by its structure (e.g., leaf data types, irrelevance of ordering of key-value pairs).

The proposed framework solves this in a very general way. This is demonstrated on a range of **uncurated** datasets, modifying *only* the path to the input data. In the `Device ID` challenge (CSP, 2019) hosted by `kaggle.com`, the samples originate from a network scanning tool. In `EMBER` (Anderson and Roth, 2018), the samples were produced by a binary file analyzer. `Mutagenesis` (Debnath et al., 1991) describes molecules trialed for mutagenicity on *Salmonella typhimurium*. Table 1 shows that the default setting of our framework, where the JSON embedding is followed by a simple feed-forward classification network, reaches a very good performance off-the-shelf (Default), while further tuning (Tunned) allows reaching the performance of competing approaches (Comp.) taken from CSP (2019) and Loi et al. (2021). Experimental details can be found at `https://github.com/CTUAvastLab/JsonGrinderExamples`. Woof and Chen (2020) also describe a framework for hmil data, but, according to limited comparison therein `JsonGrinder.jl` performs better.

Dataset	Sample Size	Accuracy Default	Tunned	Comp.
`Device ID`	0.1k-0.3M	0.935	0.971	0.967
`EMBER 2018`	3k-6M	**0.951**	0.968	0.969
`Mutagenesis`	4k-8k	0.886	0.909	0.912

Table 1: Accuracy of HMILnet with parameters from tutorial (Default), with tuned hyperparameters (Tuned), and that of SOTA solution (Comp.).

2. Background on Hierarchical Multiple Instance Learning

The set of all possible HMIL data samples, $\mathcal{H}$, is defined recursively. Any data type that can be conveniently represented as a fixed-size vector (i.e., integer, float, string categorical value) is an **atomic** HMIL sample from a set $\mathcal{A} \subseteq \mathcal{H}$. More complex HMIL samples are created using two constructions: **sets** – $\{x_1, x_2, \ldots, x_n\} \in \mathcal{H}$ for $x_i \in \mathcal{H}$; and **dictionaries** – $\{(k_i, v_i) | i \in 1 \ldots k\} \in \mathcal{H}$ for $k_i \in \mathcal{A}, v_i \in \mathcal{H}$. Keys k_i in the dictionaries are identifiers of properties with a semantic meaning (e.g., `mac`, `ip`, `services`) rather than plain carriers of information. In other words complex HMIL samples contain other HMIL samples as children.

It is common to assume that samples in one dataset obey a fixed **schema**, which means that if data in a particular set are atoms, they are of the same type and if they are more complex samples, they follow the same sub-schema. The same should hold for values under a specific dictionary key in

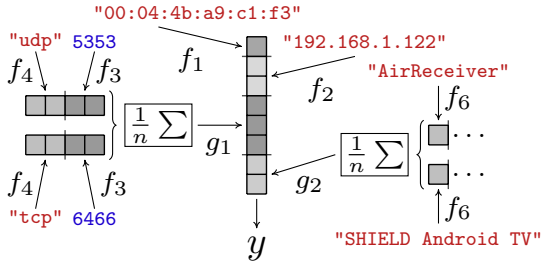

Fig. 2: A sketch of a suitable model for processing the document in Figure 1.

different samples. These assumptions are not necessary for our framework, but they are needed for the generalization to unseen samples. Some data formats enforce a schema, e.g. ProtocolBuffer and to some extent XML, otherwise the schema can be derived automatically from a dataset.

3. Overview and Design

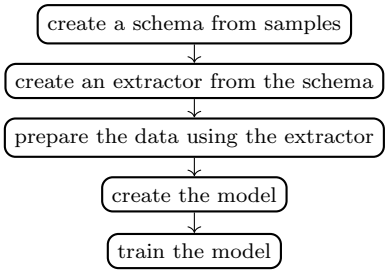

Fig. 3: Steps to create a model.

The key idea of processing HMIL data is creating a hierarchy of trainable embeddings, which gradually project atoms, sets, and dictionaries to fixed-sized vectors (see Pevný and Kovařík (2019) for the extension of the universal approximation theorem). By knowing that child data-nodes are always projected by the child-embeddings to vectors, the embeddings can be arbitrarily nested according to the structure of data. For computational efficiency, once data are converted into internal structures, they are packed to continuous tensors.

While the model for given HMIL data can be constructed manually from primitives, doing so is tedious and prone to errors. Therefore `JsonGrinder.jl` automatizes this process *without* sacrificing the flexibility. Models are constructed in five steps as shown in Figure 3[1] and briefly described below. In the following walkthrough, it is assumed that `jsons` is an array of parsed JSON documents.

Step 1 Create a **schema** of a given dataset consisting of a set of `jsons`, using the function `sch = JsonGrinder.schema(jsons)`. The returned structure, `sch`, contains basic statistics at the nodes within the data, e.g., types nodes (dictionary, array, leaf), how often is a particular element present, the distribution of lengths of lists at a specific position, the distribution of leaf values, and names of the keys of the dictionaries. `sch` can be visualized in HTML, which helps to understand the data.

Step 2 The schema facilitates the creation of an **extractor**, converting raw JSON data to internal structures derived from `AbstractDataNode`s. JSON lists (e.g., `services` in Fig. 1) are converted into `BagNode`s[2] and JSON dictionaries (elements of `upnp` in Fig. 1) are mapped to `ProductNode`s. We acknowledge that there are many ways to represent JSON leaves and the flexibility of their representation is preserved. By default, `JsonGrinder.jl` represents numbers directly, diverse collections of strings as n-gram histograms, and small collections of unique values as one-hot encoded categorical variables. The extractor can be created automatically from a schema as `ex = JsonGrinder.suggestextractor (sch)`, which uses heuristics to decide how to represent individual leaves. If the default extractors are not satisfactory, they can be easily replaced by custom implementations.

1. The complete example is available at `https://github.com/CTUAvastLab/JsonGrinder.jl/blob/master/examples/mutagenesis.jl`. The code building the model consists of 25 lines of code, the rest are mostly comments.
2. This ignores the information contained in the list's ordering, but results in much more computationally efficient training. Support for sequences can be achieved by recurrent neural networks or transformers as shown in one of the examples in `Mill.jl` , but this never achieved performance gains worth the computational cost in our experiments.

Step 3 Use the extractor, `ex`, to convert raw JSONs into internal structures using, e.g, `map` function as `dss = map(ex, jsons)`.

Step 4 Define a neural network model reflecting the schema. For the basic functionality, three types of nodes are sufficient. `ArrayNode` is the data node for atomic data and the corresponding to `ArrayModel` wraping a trainable function, e.g., a feed-forward neural networks (FNN). `BagNode` for sets and the corresponding `BagModel` implements various permutation invariant aggregation functions (a concatenation of coordinate-wise mean and maximum seems to be most effective in practice). `ProductNode` for dictionaries and the corresponding `ProductModel` containing a trainable function for each key. It applies them to the corresponding values, concatenates the outputs, and executes an additional trainable function on the concatenation. The model can be created automatically from the schema, `sch`, and the extractor, `ex`, as `model = JsonGrinder.reflectinmodel(sch, ex)`. The creation of the model is fully customizable, allowing to insert a particular FNNs and an aggregation function at each location. `model(ex(json))` projects a single `json` to a vector.

Step 5 Train and then use the model as any other model constructed by adopting the `Flux.jl` library and arbitrary associated libraries facilitating data handling.

The model handles *missing* data (e.g., missing keys in dictionaries) that can be present at all levels of the structure. Missing atomic value is expressed as a `missing` value (a feature of Julia) at the level of atomic values. During the inference, such values are replaced by trainable imputations that are unique for each node.

Integration with the ecosystem

The framework is written in the Julia language (Bezanson et al., 2017), and it is fully integrated with the Julia ecosystem. It uses `Flux.jl` for the implementation of neural networks and allows to use any automatic differentiation engine interfacing with `ChainRulesCore.jl`. Extracted JSON documents can be freely concatenated and divided, which facilitates the creation of minibatches during the training. `JsonGrinder.jl` is registered and can be added by typing `Pkg.add("JsonGrinder")` command. For Python users who want to use the library, we provide an example notebook demonstrating the interface.

4. Conclusion

`JsonGrinder.jl` facilitates the automated creation of models from HMIL data, which despite being ubiquitous in the industry are rarely considered in the ML literature. The library is flexible, extensible, and well-integrated into the Julia ecosystem, allowing to benefit from its improvement. The authors have used it in practical applications on large problems containing 10^8 samples of size up to 1GB each, frequently achieving better performance than with hand-designed features. We are not aware of any other software package that would allow the processing of JSON data without feature engineering, and therefore we consider the library to be an essential contribution to automating ML.

References

Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi: 10.1137/141000671.

CSP. Device identification challenge. `https://www.kaggle.com/c/cybersecprague2019-challenge`, 2019. Accessed: 2021-01-18.

A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2):786–797, 1991. ISSN 0022-2623. doi: 10.1021/jm00106a046.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, 2019.

Thomas G Dietterich, Richard H Lathrop, and Tomás Lozano-Pérez. Solving the multiple instance problem with axis-parallel rectangles. *Artificial intelligence*, 89(1-2):31–71, 1997.

Sadayuki Furuhashi. Messagepack, 2010. URL `https://msgpack.org//`. Accessed: 2021-01-18.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017.

Nicola Loi, Claudio Borile, and Daniele Ucci. Towards an automated pipeline for detecting and classifying malware through machine learning. *arXiv preprint arXiv:2106.05625*, 2021.

Tomáš Pevný and Vojtěch Kovařík. Approximation capability of neural networks on spaces of probability measures and tree-structured domains. *arXiv preprint arXiv:1906.00764*, 2019.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008. URL `http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html`.

William Woof and Ke Chen. A framework for end-to-end learning on semantic tree-structured data. *arXiv preprint arXiv:2002.05707*, 2020.