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

Anonymous[1]

[1]Anonymous Institution

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

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).


```
{ "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."]}
```


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

Dataset	Sample Size	Accuracy Default	Tunned	Comp.
`Device ID`	0.1k-0.3M	0.937	0.961	0.967
`EMBER 2018`	3k-6M	0.954	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.).

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.

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 **atom**ic 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.

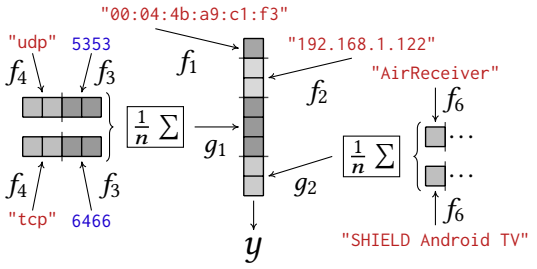

Figure 2: A sketch of a suitable model for processing the document in Figure 1.

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 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

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.

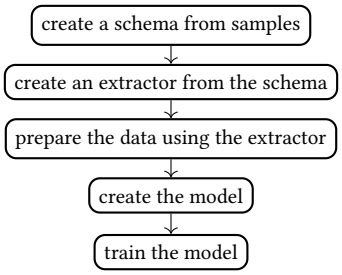

Figure 3: Steps to create a model.

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 `AbstractDataNodes`. JSON lists (e.g., `services` in Fig. 1) are converted into `BagNodes`[2] and JSON dictionaries (elements of `upnp` in Fig. 1) are mapped to `ProductNodes`. 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.

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.

[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 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

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

Bezanson, J., Edelman, A., Karpinski, S., and Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98.

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

Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. (1991). 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.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). 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.

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

Furuhashi, S. (2010). Messagepack. Accessed: 2021-01-18.

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

Loi, N., Borile, C., and Ucci, D. (2021). Towards an automated pipeline for detecting and classifying malware through machine learning. *arXiv preprint arXiv:2106.05625*.

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

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676):354–359.

Varda, K. (2008). Protocol buffers: Google's data interchange format. Technical report, Google.

Woof, W. and Chen, K. (2020). A framework for end-to-end learning on semantic tree-structured data. *arXiv preprint arXiv:2002.05707*.

5 Broader Impact Statement

The above presented `JsonGrinder.jl` simplifies use of machine learning on data stored in the hierarchical format, which is most of data exchanged on the internet. It therefore simplifies application to domains, which has been previously difficult due to the need to design features projecting hierarchical formats to tensors of fixed size needed by most machine learning toolkits. Authors therefore believe that `JsonGrinder.jl` democratizes the use of machine learning. This democratication can be potentially dangerouse. A naïve user might easily create a biassed classifier or classifier relying on non-informative features without users being aware of it. Authors are therefore developing a companion library explaining the decisions of HMILnet models, which would simplify identification of these problems.

6 Submission Checklist

1. For all authors...

   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes] We believe the claims to be accurate.

   (b) Did you describe the limitations of your work? [Yes] This paper is mainly about implementation of the framework theoretically proposed in Pevný and Kovařík (2019). Due to the page limit, we desribed limitations of the implementation briefly and they are fully described in the documentation. The theoretical assumptions on which the mathematical aparatus is built is described in Pevný and Kovařík (2019).

   (c) Did you discuss any potential negative societal impacts of your work? [No] We are not aware of negative societal impacts except the democratization of ML, which is discussed above in the *Broader impact statement*.

   (d) Have you read the ethics author's and review guidelines and ensured that your paper conforms to them? `https://automl.cc/ethics-accessibility/` [Yes] We have read them.

2. If you are including theoretical results...

   (a) Did you state the full set of assumptions of all theoretical results? [N/A] This paper describes a library built on top of theoretical foundations published in Pevný and Kovařík (2019). The Pevný and Kovařík (2019) clearly states all theorerical assumptions.

(b) Did you include complete proofs of all theoretical results? [N/A] The paper Pevný and Kovařík (2019) contains all proofs.

3. If you ran experiments...

(a) Did you include the code, data, and instructions needed to reproduce the main experimental results, including all requirements (e.g., `requirements.txt` with explicit version), an instructive `README` with installation, and execution commands (either in the supplemental material or as a URL)? [Yes] We have created a repository `https://github.com/CTUAvastLab/JsonGrinderExamples` which contains code and download the relevat datatasets if available. One dataset, DeviceId, is not available anymore due to issues of privacy. We have used it in the comparison, since we used it during development of the library.

(b) Did you include the raw results of running the given instructions on the given code and data? [No] We do not provide raw results, as the scripts were designed to output directly the reported numbers.

(c) Did you include scripts and commands that can be used to generate the figures and tables in your paper based on the raw results of the code, data, and instructions given? [No] Figures in the paper are mostly illustrative. There is one table with results, which we copied manually to the manuscript.

(d) Did you ensure sufficient code quality such that your code can be safely executed and the code is properly documented? [Yes] We did our best to make the code for experiments clean, as it was designed as an example of how the library can be used.

(e) Did you specify all the training details (e.g., data splits, pre-processing, search spaces, fixed hyperparameter settings, and how they were chosen)? [Yes] It is all in the scripts.

(f) Did you ensure that you compared different methods (including your own) exactly on the same benchmarks, including the same datasets, search space, code for training and hyperparameters for that code? [Yes] We did our best.

(g) Did you run ablation studies to assess the impact of different components of your approach? [N/A] We are not aware of space for ablation.

(h) Did you use the same evaluation protocol for the methods being compared? [Yes] The evaluation protocol was fixed for methods implemented by us. In all cases, we have preserved the split to train and test data.

(i) Did you compare performance over time? [No] The datasets were not designed to measure how the performance degrade over time.

(j) Did you perform multiple runs of your experiments and report random seeds? [No] We have not repeated the experiments, since the training and testing split of datasets was fixed. By repetition, we would measure sensitivity to initialization of weights, which in case of supervised training is usually low.

(k) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [N/A] Since we did not repeat experiments, we could not draw error bars.

(l) Did you use tabular or surrogate benchmarks for in-depth evaluations? [N/A] No tabular or surrogate banchmarks were used.

(m) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [No] The experiments were used to demonstrate the presented library. We have therefore not counted the compute time.

(n) Did you report how you tuned hyperparameters, and what time and resources this required (if they were not automatically tuned by your AutoML method, e.g. in a NAS approach; and also hyperparameters of your own method)? [Yes] All tuning is in the scripts.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...

   (a) If your work uses existing assets, did you cite the creators? [Yes] We have cited the original creators.

   (b) Did you mention the license of the assets? [No] We do not mention the licenses as we expect this information to be available at the publications listing the datasets.

   (c) Did you include any new assets either in the supplemental material or as a URL? [No] We do not publish new assets.

   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [No] We expect this not to be needed, since the data were freely available at the time of writing. The exception was a DeviceID dataset, where we had permission from Avast to use it.

   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [No] In case of DeviceID datasets, there might be a potential of privacy attacks, since the exact configuration of network devices might be specific to identify a deployment site. Because of this, the data are not publicly available anymore.

5. If you used crowdsourcing or conducted research with human subjects...

   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A] We did not used crowdsourcing during the research.

   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A] We did not used crowdsourcing during the research.

   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A] We did not used crowdsourcing during the research.