Using Domain Specific Languages for Deep Learning Code Generation

A. Rossa & N. Dethlefs

The School of Engineering and Computer Science, University of Hull

Introduction

In the recent years, Deep Learning has seen a remarkable amount of attention both from the scientific community and from the wider public. A lot of this attention can be attributed to the excellent results which this technology achieved in areas such as computer vision and object recognition, but also natural language processing and many others. To efficiently use the power of Deep Learning, however, one has to overcome the inherent complexity of the neural network models. By using Domain Specific Languages (DSLs) to abstract away this complexity, the barrier to entry can be significantly lowered and these powerful models can be easily used by domain specialists regardless of their programming skills, while, at the same time, by using the code generation approach, much of the room for fine-tuning of these models is left for the more professional users to take the advantage of.

Domain Specific Languages

DSLs are lightweight programming languages that are expressive uniquely over the specific features of programs in a given domain [4]. Their usefulness lies in creating a simple and easy-to-use interface for a particular task. This can be used by domain specialists with little coding skills as a way to use powerful software tools for work in their domain, but it can also be useful to professional programmers as a way of saving time and energy when dealing with trivial tasks.
In our work we have focused on constructing a Python internal DSL. This way, our DSL becomes a subset of the Python language and inherits many of its syntactical and semantic properties. That ties in well with the aims of our DSL, namely the code generation, since the generated code is Python as well.

Abstract Syntax Trees and ANTLR

Abstract Syntax Trees (ASTs) are widely used in programming language compilers. They are a popular choice because of their ability to efficiently represent syntactic constructs and implicitly representing relationships by means of the tree structure.

```
json_object
   : '{' json_pair (',' json_pair)* '}' ','?
   | '{' '}' ','?
   ;
json_array
   : '['json_value (',' json_value)']' ','?
   | '[' ']' ','?
   ;

json_pair: JSON_STRING ':' json_value;
json_value
   : JSON_STRING
   | JSON_NUMBER
   | json_object //recursive call for nested objects
   | json_array // recursive call for nested arrays
   | 'true' // different to the Python boolean tokens
   | 'false'
   | 'null'
   ;
```

Figure 1: Part of the ANTLR4 DEFIne grammar used by the DSL.

ANTLR [3] is a mature tool for lexing and parsing grammars of the EBNF (Extended Backus-Naur) form. Using custom grammar for our DSL (part of which can be seen in Figure 1) and running it through the ANTLR tool, we were able to obtain an AST representation of a given DSL code. In Figure 2 you can see a selected part of an AST generated by ANTLR from our DSL.

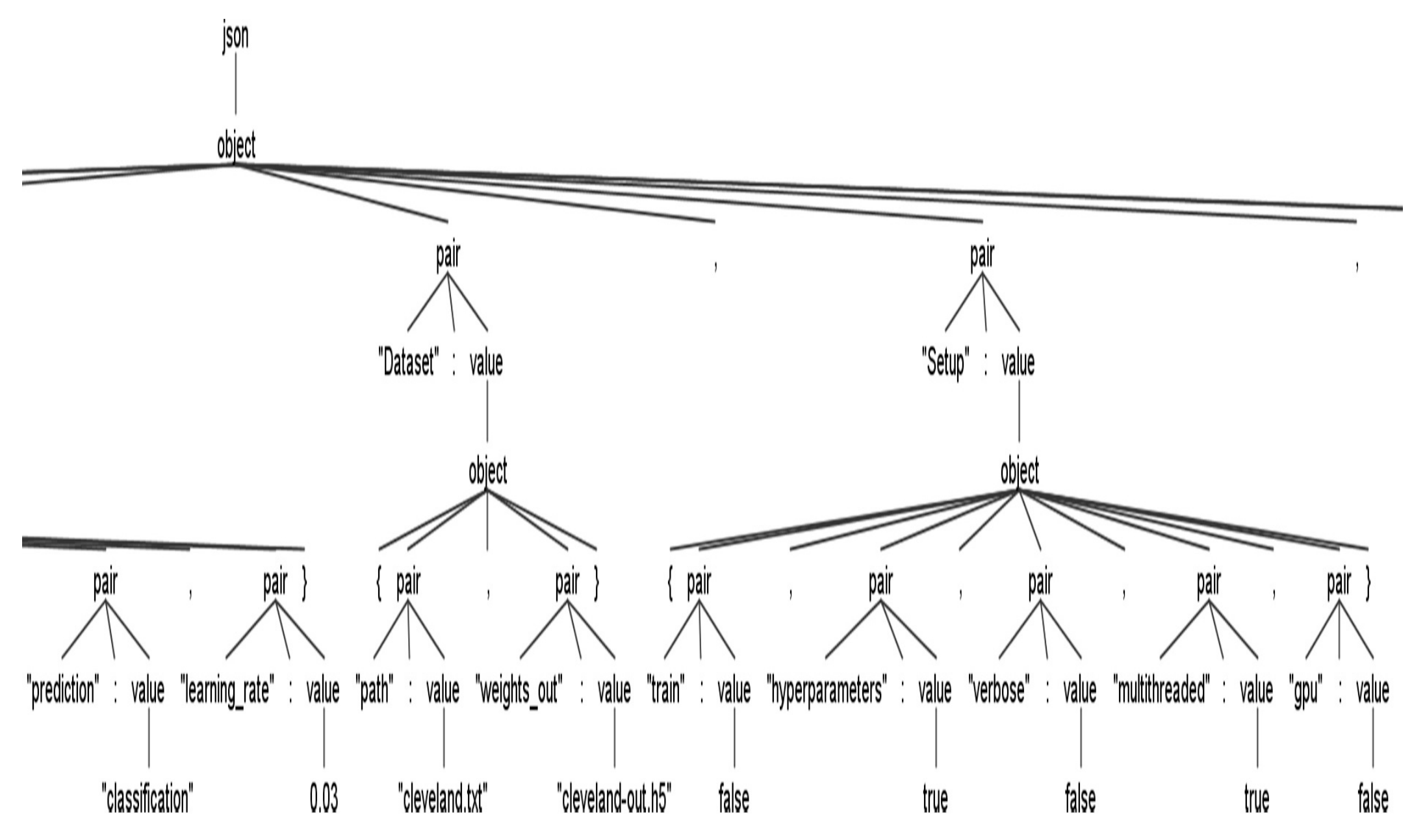

Figure 2: Part of the ANTLR generated AST showing parsed JSON configuration file.

The AST has been produced, in this instance, from a JSON configuration file, a part of which you can see in the Figure 3 below. As you can see, the AST organizes the JSON objects and pairs of values into a tree-like structure. By organizing children nodes according to the grammar rules, the structure itself carries semantic meaning and by traversing it in the right way, we are able to use this for the code generation.

```
{
        "dataset":{
                "path": "cleveland.txt",
                "weights_out": "cleveland-out.h5"
        },
        "setup":{
                "train": false,
                "hyperparameters": true,
                "verbose": false,
                "multithreaded": true,
                "gpu": false
        }
}
```

Figure 3: JSON file that the AST has been generated from.

We do this by means of walking the tree with our custom Listener class which extends the one that was automatically generated by ANTLR. The Listener walks the tree and upon visiting each relevant node, an action is performed. By performing the right action at every step, the resulting Deep Learning source code is produced.
The whole process is context sensitive since the AST provides us with information about the big picture. Therefore, different actions can be performed based on where in the tree the Listener is.

Code Generation

Among the benefits of the automatic code generation is the increased effectiveness of complex software production by reducing the cost and time associated with the coding effort [2].
Complex neural networks can be constructed using only a couple of lines of code. In Figure 4 you can see the code that was generated by our DSL. The code is sitting on top of a popular machine learning framework, Keras, which, at the moment, all of the DSL compiles to. Since the generated code is a full-blown implementation of the neural network in a given machine learning framework, there is no performance penalty incurred when using the DSL instead of just writing the code by hand.

```
self.model = Sequential()
for _ in range(self.layers):
self.model.add(Dense(self.hidden_size, input_dim=max_len,
                    kernel_initializer=self.init_mode,
                    activation=self.activation1))
self.model.add(Dropout(self.dropout_rate))

if self.prediction == 'regression':
self.model.add(Dense(1, kernel_initializer=self.init_mode,
                    activation=self.activation2))
else:
self.model.add(Dense(output, kernel_initializer=self.init_mode,
                    activation=self.activation2))


self.model.compile(loss=self.loss, optimizer=self.optimizer,
                  metrics=self.eval_metrics)


print(self.model.summary())
```

Figure 4: A look at the Keras code generated by the DSL. It shows Deep Learning model being built based on the provided parameters.

As mentioned previously, the code generation part is executed via Listener object listening while the AST is being walked and the code is generated upon walker encountering a rule that triggers an action in the Listener. The DSL handles non-DSL code - such as print() statements and such - in the DSL source file by identifying it and just passing it along with the generated code and, if viable, grouping it together.

Implementation in DEFIne DSL

DEFIne [1] is a Deep Learning DSL created at the University of Hull. It aims to facilitate work with neural networks by acting as an easy to use interface between the user and the underlying Deep Learning libraries. It offers two ways of generating the source code, either by using configuration files or by using the custom DSL. It significantly reduces the number of lines of code needed from the user to create a Deep Learning model. The produced code can either run straight after the generation or it can be further modified. One of the particular use cases for the DSL is therefore as a rapid prototyping tool that assists an experienced user with the tedious minutiae, offering them more time to spend on the actual Deep Learning model optimization.

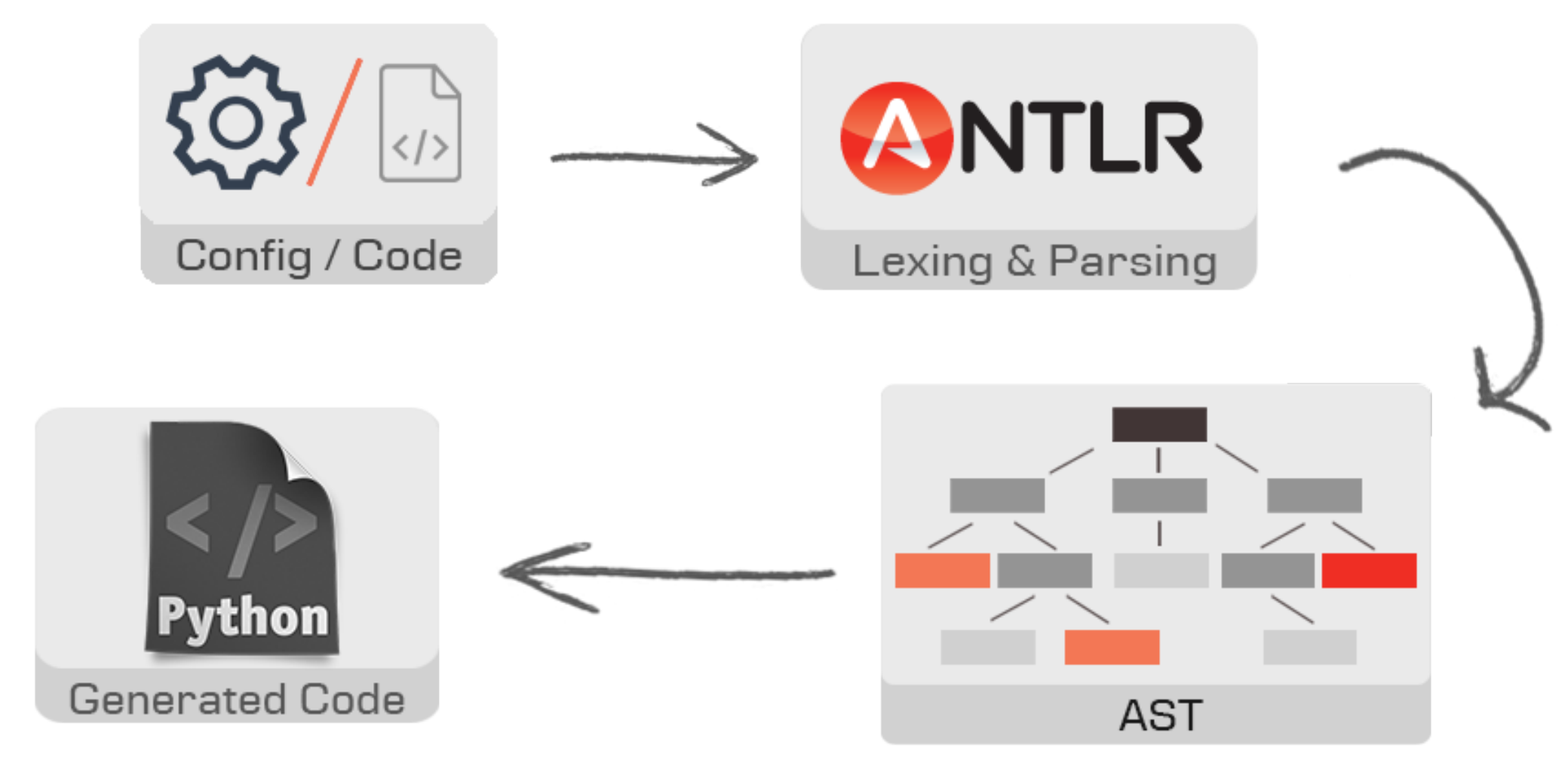

Figure 5: Shows the flow of execution of the DEFIne DSL, from initial DSL code/config file to generated source code

DEFIne implements the aforementioned in a single Python package. The user provides the dataset and uses DSL code to guide the code generation. Afterwards, by using ANTLR, the AST is produced and by walking the tree, code is generated. The code is output as a stand-alone Python script. The AST can be used for a visual overview of the intermediate representation and basic error checking mechanisms are in place to warn user about possible problems with the code. The whole execution flow can be seen in Figure 5.

DEFIne is a work in progress and there are many possible extensions to be implemented in the future. These are things like the capability to generate Deep Learning code for Convolutional Neural Networks, adding automatical hyperparameter substitution if some are omitted by the user or generating code in lower-level machine learning frameworks such as TensorFlow.

Conclusion

By using Python internal DSL in conjunction with Abstract Syntax Trees used as a tool for generation of Deep Learning code, we were able to significantly reduce the effort required from the user to successfully apply various Deep Learning models to the problems in their domain. The generated code can be either run directly, reducing the whole process of using Deep Learning models to a couple of lines of code in the DSL, or it can be generated for the user and used as a baseline for rapid prototyping. This approach is easily extended with new capabilities, such as optimization on the Intermediate Representation level, and does not require any compromises when it comes to flexibility and efficiency.

References

[1] Dethlefs, N. and K. Hawick (2017). Define: A fluent interface dsl for deep learning applications. Austin, TX, USA.

[2] Kornecki, A. J. and S. Johri (2006). Automatic code generation: Model ? code semantic consistency. Volume 1 of *Software Engineering Research and Practice 2006*, Las Vegas, NV, USA, pp. 191–197.

[3] Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2 ed.). Pragmatic Bookshelf.

[4] Thibault, S. A., R. Marlet, and C. Consel (1999). Domain-specific languages: From design to implementation application to video device drivers generation. *IEEE Transactions on Software Engineering 3*, 363–377.