The use of Recurrent Neural Networks in assisting coders

Gabriel Bastard[1], Robin Beaudot[2] and Alexandre Lenoir[3]

ABSTRACT

Recurrent Neural Networks (RNN) are very powerful sequence models that have been able to accomplish impressive tasks and Long Short-Term Memory (LSTM) have yet improved the quality that can be reached using those models. In recent years, they have become the state-of-the-art models for a variety of Machine Learning problems. In this work, we aim at using those models to create an Artificial Intelligence (AI) that can assist coders in their work, by predicting the right piece of code at the right time. We implemented the different models we trained in a plug-in for an integrated development environment, so that it could directly be used by coders.

I. INTRODUCTION

This paper aims at detailing our approach in using RNN for assisting coders. The idea is simple: use the code file as an input to predict what will be next. Our vision is that this could help automatize a lot of coders' daily redundant and boring tasks, by building an artificial intelligence that can assist coders in a smart way. The eventual goal is to enable coders to focus on the more interesting aspects of their jobs: design, engineering, algorithmic performance etc.

II. RELATED WORK

A. General

RNNs are increasingly becoming the state-of-the-art in many Artificial Intelligence tasks. A big limitation of classical Neural Networks is that they only accept fixed-sized vectors as input and output. However, RNN can operate over time-series data (e.g., sequence of characters) by their iterative nature, which offer them more power and flexibility. So whenever the dimensionality of the input and output is not fixed, RNNs are often the way to go. But even if the inputs and outputs are fixed, RNNs are still often more powerful than classical neural networks, in the sense that it is always possible to process the data in a sequential manner. Some recent research has even proven very good efficiency on more classical Deep Learning (DL) tasks (typically taking in fixed array size) such as image recognition [1]. This is mostly explained by the fact that, thanks to their structure, RNNs are not limited to a fixed number of computational steps, and can then elaborate more complex models than classical neural networks. Recently, important improvements were made with RNNs in translation [2, 3], image captioning [4, 5] or speech recognition [6, 7] thanks to the development of technologies such as Long Short Term Memory (LSTM) cells, attention mechanisms and others sharply improving RNN's performance. In many ways, RNNs carry the promise of more and more human-like AI systems.

B. Examples

Our project has been greatly inspired by the excellent "The Unreasonable Effectiveness of Recurrent Neural Network" article by Andrej Karpathy [8]. Some examples of what RNNs are capable of are detailed in this article. From conceiving brand new Paul Graham writings to inventing new baby names, A. Karpathy illustrates well what RNNs are good at: getting meaning, establishing rules, and potentially creating new things with respect to these rules, inviting imagination and creativity to AI's table. However, in our use-case, we do not need creativity, as we want the code to be perfectly answering the need of the coder and not wandering around with new variable names or untraditional coding syntaxes. Still, we needed

[1]G. Bastard is a fourth year student at Ecole Polytechnique and part of the Entrepreneurship and Innovation master program

[2]R. Beaudot is a Suparo/Polytechnique student part of the Entrepreneurship and Innovation master program

[3]A. Lenoir is a fourth year student at Ecole Polytechnique and part of the Entrepreneurship and Innovation master program

RNNs' great generalization power and their flexibility towards input length. Plus, regarding the generation of code, RNNs are particularly good at modeling syntax aspects like parenthesis pairing or indentation [9].

III. IMPLEMENTATION

A. Input and preprocessing steps

We have chosen to focus on predicting python language syntax. This choice was arbitrary and grounded by the facts that we all knew this language better than any other and that it is becoming a more and more significant language, especially regarding data science. In order to have coherence in the data set and to be able to predict more accurately, we only took python files that dealt with data analysis. Our intuition was that the RNN would get more meaning out of files dealing with the same topic, especially as we did not have very large inputs. The data thus contains many scikit libraries and Machine Learning tutorials using those libraries. We gathered most of the code from different repositories on Github and after having concatenated these files, we had to get rid of some parts in order to clean up the input:

- We removed all the comments in the files. We wanted the RNN to learn the logic behind coding, not behind commenting (which relates more to writing and did not interest us in the current context).
- We removed some really big dictionaries (more than a hundred lines) that were defined in the libraries. We thought it would lead to a better learning of the model, again because these dictionaries did not involve direct coding logic but rather arbitrary characters, names.

We were then able to train our model on a dataset of about 130,000 lines of good quality python code.

B. Technology used

At first, we coded the RNN using Tensorflow, an open source software library for Machine Learning [10]. Later, we realized that Keras would help in building validation, test and hyper-parameter optimization features with its higher-level flow so we reimplemented the RNN using Keras. However in doing so, we lost the ability to look at things (i.e nodes/cells) at a very granular level. That is why we will not present any result linked to cell activation.

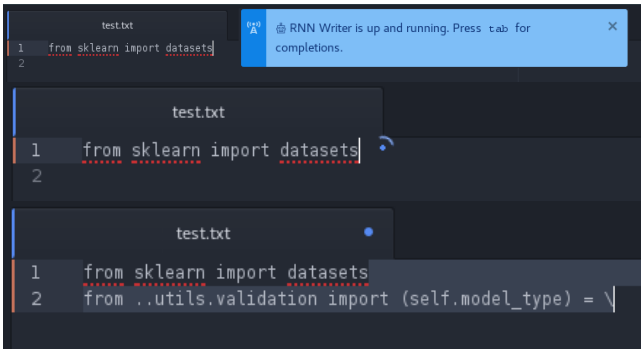

Fig. 1. Screenshots of the plugin in action: once the plugin is active, the sampling method is called upon pressing the Tab Key and the file is filled with the server's response

C. Plug-in logic

Besides the RNN logic that we have implemented and improved over time as we will discuss later, we tried to find a way to use our model in the easiest way possible. We decided to use a plug-in that could work with a text editor, so that the technology could directly be used while coding. We adapted an ATOM (a text editor developed by Github) plug-in so that it could illustrate our work. The plug-in works in a simple way : a user starts to type some code in the text editor, with the plug-in running in the background. Whenever the user presses the Tab Key of his keyboard, the plug-in calls a server on which the model is saved. This action automatically sends the so far written text as an input, which is fed to our model. ATOM is then returned the result of the server call, and prints the new RNN-generated text into the file. Fig. 1's screenshots illustrate this flow.

D. Server-side logic

The server is constituted of two main files: the sample file and the train file. Semantically defined, the first one logically samples an example using a trained model and the second one trains models on the data. A vanilla server file implements the server using Flask and allows the Atom plug-in to call the sample function upon pressing the Tab Key. We also used a third important file in our back-end logic: the hypertuning file. Used to tune

the parameters of our models, we will discuss its use later in this paper.

IV. Optimization of the prediction

During our research, we have developed more and more complex models in order to predict more and more accurate code that would best fit the coders' work. Let us now take you through the different optimizations we have led, and show you how the results evolved over time.

A. Basic RNN implementation

The first step was to implement a basic RNN model, so that we could have a first performance benchmark. We took parameters that seemed reasonable (2 layers, 256 nodes per layer) and ran our training file. With no LSTM and no hyperparameter tuning, we expected the result to be pretty bad, and it was. Here is a sample we could get using a simple RNN's model:

```
def=rd_zd8lg(f
    r2ur  sec
```

We can see that we can barely recognize python code. However, the model seems to be starting to learn a little bit of syntax, but we are very far from being able to use this as an efficient assistant for coders.

B. RNN with LSTM

The second step we took was to implement LSTM. With RNN, even if the sampling method uses previous characters to predict the next one, somehow using the "past" to predict the "future", we expected the memory to be too short-term. We suspected that things like imports or long dependencies would not be understood and taken into account at prediction time. We hoped the LSTM would improve these memory shortcomings. Indeed, LSTM cells have a much more complex memory mechanism and use input/output/forget gates [11]. For instance, when the input gate is "closed", the memory is not overwritten upon new incoming information, and this results in possibly more long-term memory. However the trade-off is that it takes much more time to train an LSTM than a RNN: there are at least 3 times more computation to do on each iteration (three internal matrices instead of one for each cell) and moreover the LSTM being a more complex model, it needs more data to be trained.

TABLE I
Hyperparameters optimization

n_modules	lr	batch_size	time per epoch (s)	loss	cat_acc
128	0.001	128	2500	1.14	0.695
128	0.1	256	2400	7.85	0.29
256	0.1	512	5400	11.38	0.29
256	0.001	256	5400	1.0098	0.73
256	0.01	512	5300	1.07	0.71
128	0.1	512	2200	5.86	0.28

C. Hyperparameters optimization

In order to obtain a powerful model, we needed to tune the model's hyperparameters. What should be the structure of the network, what should be the learning rate... ? These are the questions we tried to tackle with hyperparameters optimization. In order to achieve this optimization goal, we used hyperopt, a library built on top of Keras, which basically runs smartly chosen trainings using your defined parameters' hypercube and choses the best trained model [12]. The training and optimization took three days on a p2 AWS instance. The hyperparameters choices were:

- Number of modules : 128, 256
- Learning rate : 0.001, 0.01, 0.1
- Batch size : 128, 256, 512

We eventually achieved a 0.73 categorical accuracy with the triplet of hyperparameters (256, 0.001, 256). This does not sound like a great accuracy. However, the results are pretty promising given the relatively little computation time we were alloted on AWS. With more time, we could find a more accurate model thus decreasing the loss.

V. EXPERIMENTAL RESULTS

A. Loss and accuracy evolution

As we can observe in Fig. 2, there is very little over-fitting for this choice of hyperparameters. Indeed, training loss and validation loss are pretty close, showing that the model generalizes well over unseen examples. The apparent ceiling in validation accuracy also indicates that we have trained the model enough, and that there is no need for more training epochs. This detail allowed us to check that hyperopt was indeed stopping the training at the right time.

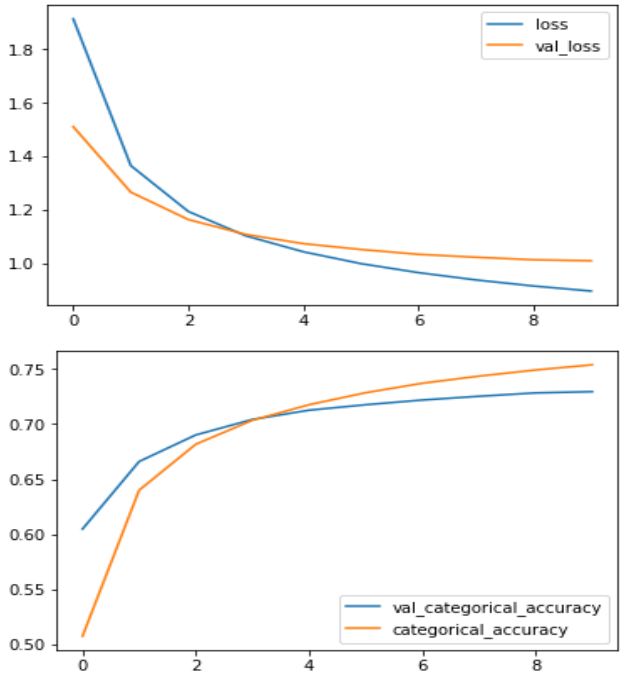

Fig. 2. Evolution of the model loss, validation loss, categorical accuracy and validation categorical accuracy during training for 256 modules, 0.001 learning rate and a batch size of 256

B. Evolution of samples during training

The graphs on accuracy (Fig. 2) are good to see if there is overfitting in the training and the evolution of the loss, however they do not help in seeing how the model really performs at different stages of the training. It is hard to imagine a metric that measures how "normal" a sample of code is. It was interesting for us to take a look at samples' evolution during training. Here is how it looks like: At first, we only observe mixed letters without any resemblance to code.

```
def=pre7d;a^X,_ar^1)7Xmal;dpl
kiense,
```

After a few iterations, the model starts to figure out the idea of spaces separating words. However we observe very long words and sometimes multiple spaces.

```
def isn;**
 inse


itse


if nsimant
```

Then after more training time, the model starts getting an idea about code syntax. In particular, it is able to close brackets.

```
import os
def culaDeviceClamedoc()
    return_toldation_pred,
    axis ==
    n_np.asarray(learner_mask,
```

At the end, we almost always have executable code, with variables that are normally named, but the code does not always seem to make sense. Plus, the model often happens to get stuck in a loop of spaces, thus only generating spaces... which is clearly not useful.

```
import os
def __init__(self, 'set')
self.__content_train_sizes
if max_iter is None:
        cdef double color =
        colors.sum(self.classes())
```

It is interesting to note that the sampling method can take up to 10 seconds to generate these results. This compromises the usefulness of this practice for coders who need a quick result.

VI. CONCLUSION

In this work, we tried to develop a smart assistant for coders that could help them with redundant tasks by auto-completing parts of the code. We used a recurrent neural network with long short-term memory to create that assistant. It was also implemented in a text editor plugin so that we could use it directly. After finding the best hyperparameters for our model, the results were promising, our model being able to predict code that fits to what the coder is typing in entry. However, the number of combination of hyperparameters that we have been able to test was limited, and we could certainly improve the quality of prediction with other hyperparameters. This is also true for the size of our dataset, that we purposely limited to a certain type of source code, but that would benefit from getting larger. Finally, we have found some limitations in the usefulness of the generated samples: the time taken by the model to sample makes this approach hard to use in practice. Would this work to be continued, it would need to focus

on use-cases where immediacy is not required and where it is not a problem to wait a few seconds before getting the result.

REFERENCES

[1] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell. Long-term recurrent convolutional networks for visual recognition and description. arXiv preprint arXiv:1411.4389, 2014

[2] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473, 2014.

[3] Minh-Thang Luong, Hieu Pham, and Christopher D Manning. 2015b. Effective approaches to attention-based neural machine translation. arXiv preprint arXiv:1508.04025

[4] Xu, K. et al. Show, attend and tell: Neural image caption generation with visual attention. In Proc. International Conference on Learning Representations http://arxiv.org/abs/1502.03044 (2015).

[5] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo. Image captioning with semantic attention. In Proc. IEEE Conf. Comp. Vis. Patt. Recogn., June 2016

[6] Dzmitry Bahdanau, Jan Chorowski, Dmitriy Serdyuk, Philemon Brakel, and Yoshua Bengio,End-to-end attention-based large vocabulary speech recognition, arXiv preprint arXiv:1508.04395, 2015.

[7] B. Shi, X. Bai, C. Yao, An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition, in: CoRR, 2015.

[8] A. Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

[9] A. Karpathy, J. Johnson, and F. Li. Visualizing and understanding recurrent networks. arXiv preprint, 1506.02078, 2015.

[10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Man, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Vigas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. Tensor-Flow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[11] S. Hochreiter and J. Schmidhuber. Long short-term memory. Neural Computation. 9, 17351780, 1997.

[12] J. S. Bergstra, D. Yamins, and D. D. Cox. Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms. In Python for Scientific Computing Conference, pages 17, 2013.