# Transliterate from Diagrams and Data

Mhajeri Mahdi and Wuilloud Jair[*]at interscript.org

May 16, 2022

## Abstract

A novel graphical and data driven approach to transliteration is presented:

Linguists are being proposed to design own transliteration strategies on a graphical editor. Our technology allows to transform their diagrams into code the help of a developer.

Thinking about transliteration as a transduction problem, we have built a generic setup allowing to train neural networks and to port them into production, in ruby.

The approach allows to build and deploy efficient transliteration strategies for any language.

## 1 Introduction

Transliteration is the process of transforming a language into another script, transforming the letters to maintain the sounds.

### 1.1 Problems

Confronted to the problem of implementing a high quality transliteration strategy for farsi, our team of linguist (mother tongue farci) and software engineer (european) was faced with multiple challenges for which we found no existing solution:

[*]wuilloud@gmail.com

**Linguist**

- Hard to convey clear visions to developer
- No visibility on implementation
- Difficulties to easily test, try, redesign, ...

**Developer**

- Hard to understand and implement strategy
- Technical initiatives potentially disruptive
- Extenuating task consisting in fixing multiple special cases, details, fixing recursive loops, etc...

**Dependencies and Challenges for Production**

- Multiple resources and mappings
- Total dependency of NLP libraries and a language (python, java, ...)
- Portability issues with other architectures, languages

### 1.2 Proposed Strategy

In order to improve this, we have developed the following strategy, that we think to be quite generic:

**Transliteration design + code generation**

1. linguist designs code diagrams
2. code diagrams are exported (into csv)
3. nodes implemented by developer
4. ⇒ generation transliteration code (from the diagram export)

**Learning of transliteration with nnets**

1. transliteration dataset generation with code
2. ⇒ training of Transformers (nnets)

**Production/Transliteration in ruby**

1. torch NNets exported to onnx
2. ⇒ run production code in ruby (or converting it to javascript [3])

We now review the above described strategy in more details. Somebody interested in a practical quickstart might very well go through our **quickstart** [1].

## 2 Transliterate from Diagrams

Our linguist worked out an extensive, original solution to combine various mappings, rules and transformations (lemmatization, , stemming, PoS tagging, ...)

Better than many description, the full architecture is available online with a diagram: transliteration strategy.

Experimenting with various approaches for this project, we decided to work with lucidchart. However, any technology allowing to create and export workflows could be used.

[1]QUICKSTART

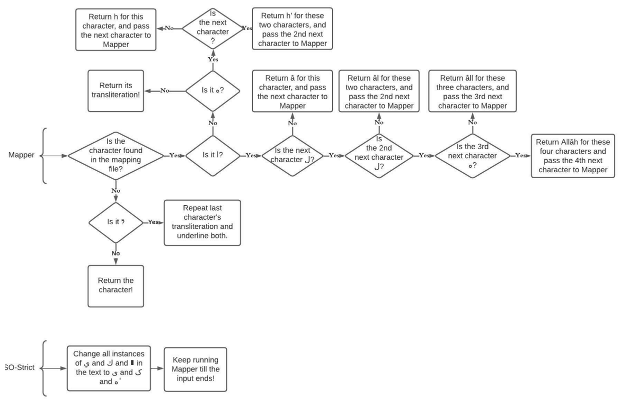
Return h for this character, and pass the next character to Mapper
No
Is the next character ?
Yes
Return h' for these two characters, and pass the 2nd next character to Mapper
Yes
Return its transliteration!
No
Is it ه?
Return â for this character, and pass the next character to Mapper
Return âl for these two characters, and pass the 2nd next character to Mapper
Return âll for these three characters, and pass the 3rd next character to Mapper
No
No
No
No
Mapper
Is the character found in the mapping file?
Yes
Is it l?
Yes
Is the next character ل?
Yes
Is the 2nd next character ل?
Yes
Is the 3rd next character ه?
Yes
Return Allāh for these four characters and pass the 4th next character to Mapper
No
Is it ّ?
Yes
Repeat last character's transliteration and underline both.
No
Return the character!
ISO-Strict
Change all instances of ي and ك and ▮ in the text to ی and ک and ه ٔ
Keep running Mapper till the input ends!

Figure 1: Basic transliteration Strategy, ISO Strict for Farsi.

### 2.1 Building code from diagrams

Once exported into csv format, diagrams can be computerized and modelled as trees, with a rule associated to each node.

The code uses a functional approach with the data "states" getting transformed as they "flow" within the graphical, logical structures. Details can be found in the original, open source Julia code[2].
Among the features that had to be created:

- Modelling of logic components as Trees
- Modelling of computation "states"
- Jumps from one logic flow to another
- Handling recursions

### 2.2 Nodes/Rules

An action to be performed is associated to each node, the action is described in plain English.

[2]learn-graph

For illustration, some examples from our diagram in Fig.1:

- node | $\gg$ **"output its transliteration!"**
- node | $\gg$ **"is it a verb?"**
- node | $\gg$ **"transliterate it using affix-handler"**

The current farsi solution is based on a design with over 150 nodes, dealing with main and edge cases. This represents a very complex solution.

### 2.3 Modularity & Flexibility of Solution

We found useful to allow for the possibility to build codes by passing the relative directory path.

This allow a flexible approach and to easily work with smaller diagrams. We hope to allow for scalability and interactivity.

### 2.4 Choice of Julia

We have chosen Julia because of the two following advantages.

#### 2.4.1 Support for python, bash, and others

Julia allows to easily run external commands, script but also integrate python, R, and C++. Examples of how to run code snippets in other languages are present in the Quickstart.

#### 2.4.2 Code Performance

Julia compiles "just in time", meaning that the code takes some instant to load and precompile all the necessary functions.

This disadvantage disappear in production where julia is notoriously fast. For instance, we found our extensive solution built from graphs to run 3 times faster than an previous python implementation.

## 3 Transliterate with NNets

We now motivate and describe transliterations realised with nnets.

### 3.1 Transliterate with transformers

Transliteration can be viewed technically as another transduction problem for which neural networks have been applied with great success.

### 3.2 Dependencies

Below, in Table.1, we start to compare the resources between a standard implementation for farsi and the production with nnets in Ruby.

| Resources for Farsi Transliteration | | |
|---|---|---|
| Implem. | Dependency | Size (Bytes) |
| Code | Mappings | 7k |
| | Library | 70M |
| | Graph | 80k |
| NNets | Onnx nnet | 85M |
| | Vocab | 45k |

Table 1: Resources requirements are comparable between a standard and nnets implementation. The ruby implementation is created from data and as such needs very few components after training.

Without the nnets approach, several complex NLP libraries would have needed to be retro-engineered and rewritten in ruby...

### 3.3 NNets training accuracy

Although the resources are comparable, we prove below that one is able to closely approximate the transliteration code with nnets if provided with (or generating) enough transliteration data.

In Fig.2, the transliteration accuracy is evaluated with word accuracy (% of correctly transliterated words) on random normalised text data,similar to the ones the nnets have seen during training.

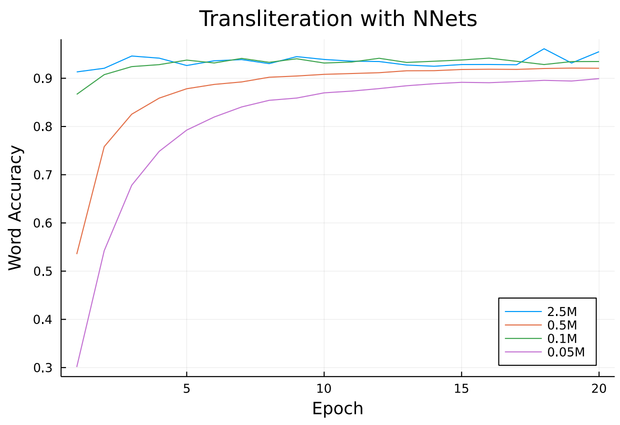

Figure 2: Replication quality for various data sizes. On a well cleaned dataset, resp. 97% and 92% of accuracy can be reached for 2.5M and 0.1M examples (after preprocessing).

### 3.4 Available Code

Training and ONNX conversion is available online[3].

[3]python-nnets-torch

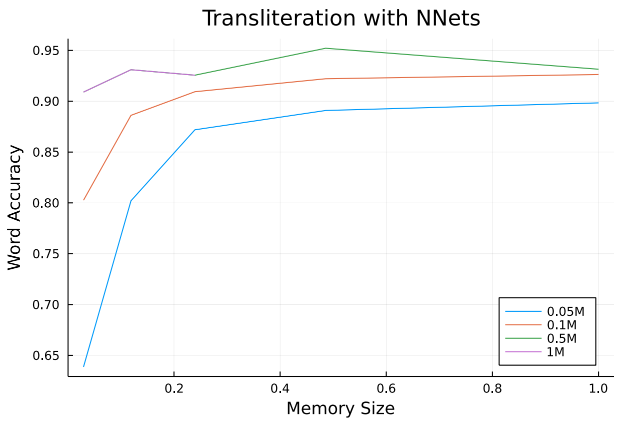

Figure 3: Replication quality for various nnet sizes.

## 4 Production in Ruby

Here again, the code is available online [4] and we have solved several problems as:

- Implementation of greedy decoding,
- Search of substrings if word unseen by algo. (word level encoding),
- decomposition of transformers into multiple sub-components.

In production, we found that various components (neural networks) of the transformers had to be exported, such as generator, tokenizers, encoder, and decoder.

They had then to be combined correctly in our native ruby code [5].

[4]transliteration-learner-from-graphs
[5]ruby lib

## 5 Benchmarking

### 5.1 Data

Most of our datasets to apply our transliteration method on were publicly available from Farsi NLP communities and Github repositories.

As a second step, transliteration data was generated by applying to it our diagrams-generated code.

We have also produced a test dataset to benchmark various transliteration algorithms. With this data, we have tried to cover many cases our rules were designed to solve.

### 5.2 Scores

| Word Accuracy | |
|---|---|
| Code | 88% |
| NNets | $\geq$ 50% $\star$ |
| Uroman[6][2] | 5% |

Table 2: Scores on a test harness with special cases that we wanted the farsi transliteration to handle and in a way our library is specialised on. $\star$ NNets getting re-evaluated.

## 6 Practical observations

- We found that empowering a linguist to be able to design and visualise complex logics was by far more successful than a first attempt separating developer/linguist workflows. This possibly because the linguist could apply is understanding/intuition directly.

- The solution (diagram to code) is very portable, visual, intuitive and creating less dependencies as software consists in simple code snippets/actions.

- Details could be improved for debugging, specifying types, expected inputs/outputs and standards, for example at the editor level (alternatives to LucidChart).

## 7 Summary and Conclusion

Thinking that the interface between software developer and linguist was one of the challenges in the development of transliteration algorithms, we have approached the problem with a graphical editor allowing a linguist to creates his own logic designs.

In a second step, transliteration is put into production after training of neural networks, allowing to bypass the usage of NLP libraries not available in ruby, but also for a more compact solution (lower or comparable memory consumption while getting rid of software dependencies).

We think that the approach or part of it can be ported to the transliteration of any other languages, especially the ones where no transliteration data is available.

### 7.1 Outlook

We hypothesize that the approach seems promising for other cases not necessarily confined to transliteration.

We are currently thinking about possible usage and improvements:

1. Alternatives to lucidchart or an plugin

2. Encourage the exchange of solutions/diagrams across users of the technology.

3. library of templates for Linguists to use?

## References

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser and Illia Polosukhin. Attention Is All You Need, 2017; arXiv:1706.03762.

[2] Hermjakob, Ulf and May, Jonathan and Knight, Kevin Out-of-the-box Universal Romanization Tool uroman, 2018; Proceedings of ACL 2018, System

[3] Blog post on interscript site Webassembly and advanced regular expressions with opal 021-06-26-webassembly-and-advanced-regular-expressions-with-opal