An API and Visual Environment to Use Neural Network to Reason about Source Code

Alexandre Bergel
Pleiad Lab, DCC, University of Chile
Chile

Paulin Melatagia
Université de Yaoundé I, UMI 209 IRD
UMMISCO-UY1, Yaoundé
Cameroon

Serge Stinckwich
Sorbonne Université, IRD, Unité de Modélisation Mathématiques et Informatique des Systèmes Complexes, UMMISCO, F-93143, Bondy, France
France

ABSTRACT

Neural networks are gaining popularity in software engineering. This paper presents a dedicated API and visual environment to train and use a neural networks on software source code related data. This short paper illustrates the API using two examples involving prediction of source code properties.

CCS CONCEPTS

• **Software and its engineering → Software system models**;

KEYWORDS

Neural network, source code, visual environment

ACM Reference Format:
Alexandre Bergel, Paulin Melatagia, and Serge Stinckwich. 2018. An API and Visual Environment to Use Neural Network to Reason about Source Code. In *Proceedings of 2nd International Conference on the Art, Science, and Engineering of Programming (<Programming'18> Companion).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3191697.3214340

1 INTRODUCTION

An *artificial neural network* is a computing system inspired by biological neural networks found in vetebrate brains [1]. Such an artificial network is a collection of connected artificial neurons. Connections between artificial neurons can transmit a signal from one to another to answer to a particular stimulus. The artificial neuron that receives a signal can process it, and then signal neurons connected to it. A neural network acquires knowledge through learning, typically from a set of inputs and expected outputs. Once properly trained for some particular problems, an artificial neural network can predict or make regression on data.

Neural networks are very popular to process arbitrary images and texts written in a natural language as they are often successful at classifying and making predictions. However, neural networks are seldom applied to software engineering tasks, in contrast to image, text, and sound processing. Numerous modeling libraries and frameworks are available to operate with neural networks. However, we believe that reducing the gap between current neural network-based models and source code entities represents a significant challenge. We provide an API that ease the manipulation of source code elements with a neural network.

Contributions. This paper describes the experience we gained in building a generic infrastructure to apply neural networks to solve some tasks involving source code manipulation. In particular, we designed a small API to make a neural network operates on any arbitrary data structure, including structural source code entities. The training of the neural network may be visually monitored using a dedicated visual environment.

Outline. We first present a small API and a visual environment to train and make prediction on any arbitrary data structure (Section 2). Second, we present an example that consists in predicting the package of a class based on the class name (Section 5). Third, we present a second example that predict whether a method has a comment or not based on the set of messages the method sends (Section 6).

2 API TO USE NEURAL NETWORK

Neural networks are conceived to operate on numerical values: feeding numerical values to a neural network produces a set of numerical output values. We have designed an API[1] to encode and decode any arbitrary non-numerical value. Consider the following example:

```
1 n := NNLang new.
2 n data: #('hello' 'bonjour' 'cow' 'tea' 'bag' 'circle' 'house' 'table' 'chair').
3 n feature: [ :w | w asArray ].
4 n expectedOutput: [ :w | w size ].
5 n epochs: 3000.
6 n train.
```

The script above uses the API we have designed to create a neural network, trained to determine the length of a string character. Line 1 instantiates an object `NNLang`, which is a wrapper of a neural network. Line 2 designates the data the network has to learn from. The data is passed as an array of words. The size of the words ranges from 3 (*e.g.,* `'cow'` has 3 letters) to 7 (`'bonjour'`). Line 3 indicates the relevant features that has to be considered for each data elements. For example, we have `'cow' asArray` produces `#($c $o $w)` an array of three characters. The words provided as data elements provided in Line 2 are made of 16 characters.

[1] We are using Pharo[2] (http://www.pharo.org) for all the examples provided in this paper.

<Programming'18> Companion, April 9–12, 2018, Nice, France

ACM ISBN 978-1-4503-5513-1/18/04…$15.00
https://doi.org/10.1145/3191697.3214340

Line 4 indicates what the expected output of each data element has to be. The block `[ :w | w size ]` takes a word data element as argument and returns the size of that element. For data set provided in Line 2, there are four different possible outputs (3, 5, 6, 7). The neural network will therefore have four different output elements, each representing a possible output value.

We do not specify the topology of the neural network, our API will therefore creates a fully-connected network with one hidden layer, with 22 neurons (number of inputs, 16, multiplied by an arbitrary factor of 1.3), and one output layer with 4 neurons (number of different outputs).

We can employ a trained network to guess the size of some words. For example, the following expressions:

```
n guess: 'tag'.
n guess: 'label'.
n guess: 'journee'.
```

correctly evaluates to 3, 5, and 7. These words are not part of the training data set, however, the neural network uses the relations learnt during the training to correctly guess the size of the words. Naturally, this simple example cannot make a guess that it has not learnt. For example, it will wrongly guess the size of the word `'papa'` since the neural network has never seen a word with 4 characters. Characters that were not part of the training words will be simply ignored. However, providing a large and varied corpus of words is able to make accurate guesses of the size of the word.

The data set provided using the keyword `data:` is a set of words in our example. The blocks to compute the `feature:` and `expectedOutput:` are evaluated to each data set element. Most of classification tasks employ a simple technique called *one hot encoding* to translate a set of labels into a binary representation, suitable for neural network feeding. Using a one-hot encoding, each of the possible 16 distincts characters is represented as an input entry of the neural network.

The next section presents the visual environment associated to this API while the following sections will use classes and methods instead of arbitrary strings.

3 VISUAL ENVIRONMENT

Neural networks are known to be difficult to configure and to give explanations of a given result (the black box problem). We have designed a set of visualization and navigation tools to navigate through the different visual projections and representation of a neural network.

Figure 1 shows the environment in which the script given in the previous section is executed. The figure shows six different visual projections, marked from **A** to **F**:

- **A** – The main code editing window is composed of two parts. The left one contains the script to be executed. The right part contains a projection of the result of the training and guessing phase. It provides some metrics about the performance and topology of the network. Some tabs are available to select a different projection, as listed below.
- **B** – One useful metric to measure the performance during the training phase is the number of failed guesses made during the training phase. The training phase uses a backprogragation mechanism, which makes a guess for each training data set element and adjusts the internal representation of the network based on how well it guessed the element. The provided curve indicates the rate of guessing during the training phase. When the curve reaches (or getting very close to) the X-axis, then the neural network is considered as trained.
- **C** – During the training phase, the backprogragation algorithm computes a delta value for each neuron. This value corresponds to the adjustment made during the training phase on the output layer.
- **D** – Gives the same information than in C, but per examples. Each curve indicates the global adjustment made by learning each example.
- **E** – Gives the topology of the neural network. In this example, its has one hidden layer, made of 22 neurons, and one output layer, made of 4 neurons.
- **F** – indicates the good guess vs wrong guess during the training phase.

4 BENCHMARKS

This section briefly presents three applications we used to run our experiments: *GeneticAlgo* is an implementation of a genetic algorithm; *Trachel* is a visual elements object model; *Roassal2* is a mapping model and set of interactive data visualizations.

Systems	NOP	NOC	NOM	LOC	% CM
GeneticAlgo	8	40	287	3,462	34.15
Trachel	10	125	1,298	7,132	22.96
Roassal2	75	753	8,383	93,623	17.27

We use the following metrics to characterize the applications: *NOP* (Number Of Packages), *NOC* (Number Of Classes), *NOM* (Number Of Methods), *LOC* (number of Lines Of Code), *%CM* (ratio of commented methods).

5 EXAMPLE: PREDICTING PACKAGE FROM CLASS NAMES

A class represents a software component for which its purpose is usually reflected in its name and in the package which the class belongs to. Studying the possible correlation between the name of a class and the package the class belongs to is the topic of this section.

The research question we would like to verify is whether the name of a class may be used to determine the package that may contain the class. Put differently: *do the classes contained in a package follow a determined naming convention?*

We answer this question by training a neural network with class names cut into camel case compound words and the package of the class as desired output. Consider the following script:

```
n := NNLang new.
n cut: 0.8.
n data: applicationClasses.
n feature: [ :cls | cls name cutWhereCamelCase ].
n expectedOutput: [ :cls | cls category ].
n epochs: 1000.
n train: 10.
```

The variable `applicationClasses` is bound to the classes of an application. The `feature:` instruction extracts the relevant feature of a class, in our case, the compound words of the class name. For example, the features of a class named `RTLineShape` are the words `RT`, `Line`, and `Shape`. The desired output is the package (called `category` in

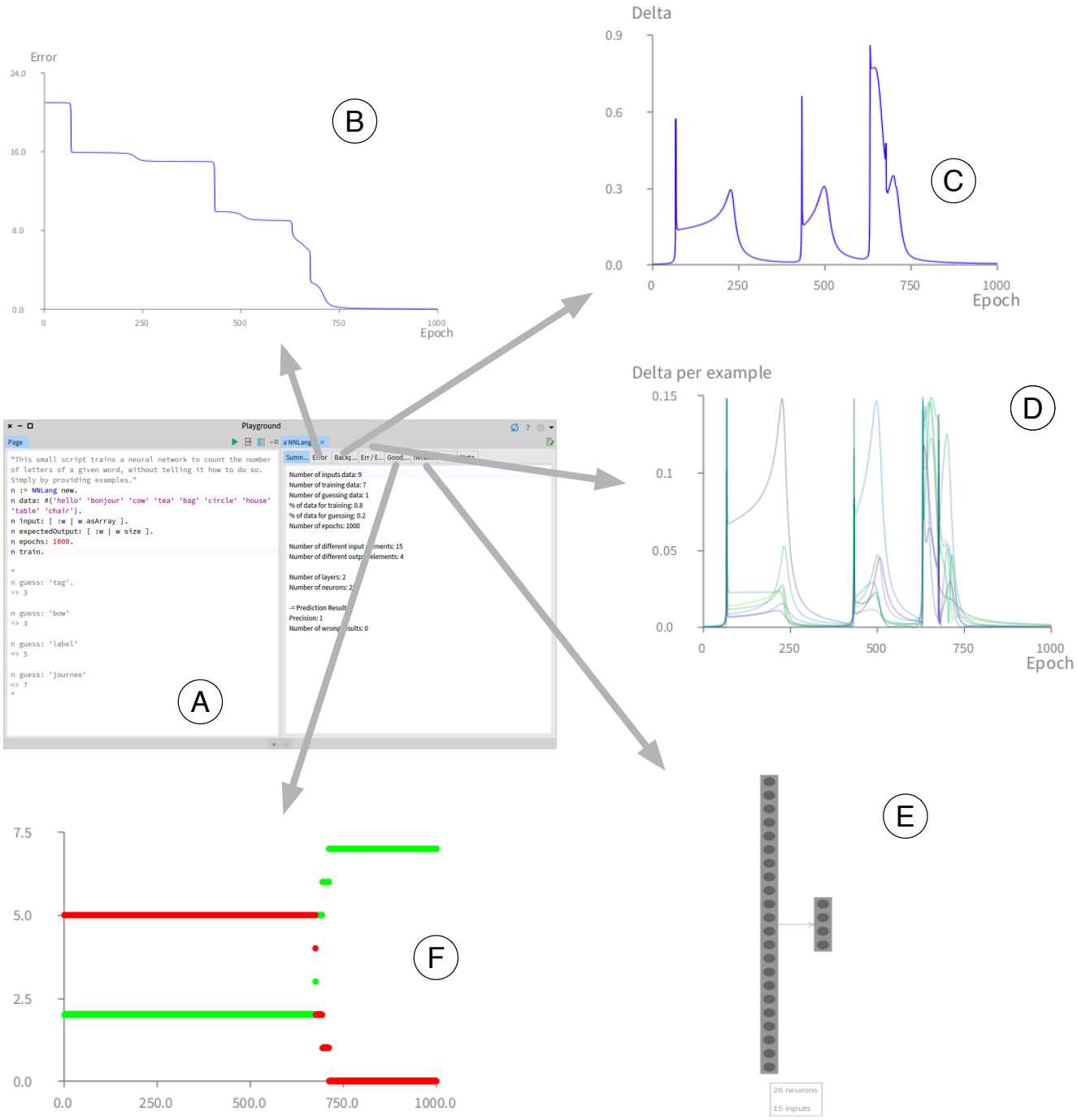

Figure 1: Live environment for programming and using a neural network

Pharo) to which the class belongs. Note that cutting a class name as camel case does not preserve the order of the compound words. From the whole set of classes, we consider 80% of `applicationClasses`, randomly picked. The remaining 20% are used to compute the accuracy of the network. The network has only one hidden layer, made of 10 neurons.

We obtained the following accuracies:

- GeneticAlgo: 14%
- Trachel: 83%
- Roassal2: 57%

The different view indicates that the global error rates (Projection C in Figure 1) and the error per example (Projection D) are reaching a value extremely close to 0. This indicates that the data are not contradictory. An example of contradiction could be a class named `ShapeLine` belongs to a package `P1` and another class named `LineShape` belongs to another package `P2`. In such a case, the network cannot properly learn due to the contradictions. However, we have not faced such a case.

Trachel has a high precision. A manual inspection of the code reveals that the code is well structured and support a strong coding convention. On the opposite, classes of GeneticAlgo do not seem to contain a particular coding convention.

6 EXAMPLE: PREDICTING THE PRESENCE OF METHOD COMMENTS

Software engineers do not always comment their methods. Within our benchmark, GeneticAlgo is the system that has the most commented methods (34.15% of the 287 methods are commented) while Roassal2 contains the fewest commented methods. We would like to explore is whether a method body can be an indicator of the presence of a comment or not.

Consider the following script:

```
n := NNLang new.
n data: applicationMethods.
n cut: 0.8.
n feature: [ :compiledMethod | compiledMethod messages ].
n expectedOutput: [ :compiledMethod | compiledMethod comment notNil ].
n train: 80.
```

The script creates a neural networks that accepts as input data the set of the method defining of the three applications we used in our benchmark. The network takes as input features the messages (*i.e.,* method call) sent by a method. The network learns from 80% of the input methods whether the method is commented or not.

We obtained the following precisions:

- GeneticAlgo: 57%
- Trachel: 51%
- Roassal2: 40%

Consider the first application, GeneticAlgo. The network correctly identifies whether a method is commented or not for 57% of the methods from the guessing set (20% of the 287 methods). If the result would be close to 34.15% (the actual ratio of commented methods), then the network would behave randomly. However, it is not since the correctly guessed ratio is much larger.

7 CONCLUSION AND FUTURE WORK

This short paper presents a simple API and visual environment to apply a neural network to verify some properties related to the source code.

As a future work, we plan to provide a way to automatically characterize the quality of the input data. Input data may be difficult to get it correctly. Based on the small example given in Section 2, if we would provide a few thousands of words long of 3 letters, then the network will always output 3, and the prediction will be high since the guessing dataset will have few occurrences of words not having 3 letters.

Acknowledgments. We thank Lam Research for partially sponsoring the work presented in this paper.

REFERENCES

[1] I. Goodfellow, Y. Bengio, A. Courville, Y. Bengio, Deep learning, MIT Press Cambridge, 2016.
[2] A. P. Black, O. Nierstrasz, S. Ducasse, D. Pollet, Pharo by example, Lulu. com, 2010.