# Torc White Paper (Extended)

**November 11, 2018**

Rani Fields

## Abstract

Torc is a format which aims to enable recurrent neural networks to read JavaScript files for classification purposes. Torc is fully open source. A torc file is a special kind of encoding for JavaScript files and is presented to the user as a CSV. When implemented, customization is encouraged but not required; a user can employ torc as-is. Due to pressing needs, torc's current focus is on security. Long-term goals include non-security use cases such as quality assurance and performance analysis.

## Preface: Where we stand today

When we peer into the world of JavaScript and recurrent neural networks, or RNNs, we find ourselves constrained to simply running neural networks in JavaScript; there appears to be no standard library which is capable of loading JavaScript into neural networks. This is of special interest since RNNs specialize in time series data and various other classes of sequential data. Thus, logic follows that RNNs are theoretically well suited to classification tasks in the domain of programming languages, and of interest, interpreted languages such as JavaScript.

## Bridging the JavaScript-RNN Gap

Loading JavaScript into RNNs is a non-trivial affair. In order to achieve our singular goal, we need to somehow translate JavaScript into some sequential format with a static number of columns while maintaining as much structural information as possible and minimizing noise. This is a daunting set of requirements but if we employ certain standard code interpretation

processes in tandem with a general mindfulness of how RNNs work, we can sufficiently achieve our goal of bringing JavaScript into the realm of RNN use cases.

## Introducing Torc

Torc is our open source bid to bridge this gap. Torc is both an interim format and the method used to generate this interim format. If a user were to open a torc file, the user would see a CSV. This is intentional- torc is fundamentally a CSV. Due to the frequency of CSVs in the machine learning world and due to the nature of our problem at hand, the CSV format filled a natural niche in how we present our re-encoded JavaScript file. When we look at torc files in depth, users will find that torc actually encodes a sequence of events alongside special metadata designed to contextualize provided information wherever possible. Through the specially engineered torc format, users can then load their JavaScript files into RNNs as they please.

## The Torc Method

The JavaScript encoding method we found which satisfies the requirements previously outlined is, just like the subject matter, non-trivial. The core concept is to take to take some JavaScript code, generate a tree based on the structure of the code, perform a specific set of modifications to the tree while collecting metadata on certain data within our tree, then perform a final walk through the tree and record important information as we see it. We then have what is effectively the results of a depth-first search on a de-noised abstract syntax tree with metadata about the different nodes provided where possible. We also provide metadata focused around other interesting entities such as text.

## The Denoising Process

The heart of torc's denoising process is best described as a data mapping method where we map variables to metadata whenever possible. This method was designed with a simple major goal in mind: the denoising process must minimize major sources of noise (text, variable names, and select tautologies) while maintaining the core structure of the code. Thus, through the denoising process, most important data should be preserved even if we must represent our data

in a new form (re-encoded via metadata). Any sources of noise (notably variables/functions and text) should be presented only as metadata; we do not reference different variables directly.

In order to denoise our abstract syntax tree, we first walk through our tree and generate basic structural data. Notably, we record the maximum depth of the current subtree for each node. We then walk over our tree to change any references to a key within each applicable object to a single object-key name. For instance, we change `varA`.`varB` to `varA.varB`. This is used for frequency information and other similar metadata. We will also record any potential variable-level tautologies at this time (re-assigned variables). We then walk over the tree once more to convert our variable-level tautologies to a single key for each group of variables while collecting basic frequency information for our final metadata. On the next iteration, we begin to substitute specific information with metadata about what we've seen thus far. Notably, variables are replaced at this stage with their metadata alternative form. Finally, we do one extra pass to actually emit any information we've collected thus far. The end result is a dataframe mixing one-hot encodings and dense data where the one-hot encodings represent node type and the dense data represents metadata about the different nodes.

## Torc's Assumptions and Feature Engineering Rules

Since torc was designed around the structure of code and not the content of code, we operated under a set of assumptions for our initial feature set. When designing new features for torc, it is highly recommended to keep these assumptions and rules in mind.

1. Assume nothing about the runtime environment. Because we cannot guarantee anything about the user's runtime environment at the time of code execution, we cannot be certain that certain objects will have certain statuses and that certain functions will operate as expected. Thus, torc features should focus on the JavaScript engine, not what happens at runtime.
2. Do not assume to know what a piece of code will do at runtime. This follows from (1).
3. Scoping-related features should be implemented with caution. This follows from (2).
4. Select your metadata features such that you can re-use the same feature for as many different structural tags as possible. In other words, you want to achieve some minimal level of uniformity in how different entities in JavaScript are presented in your torc

format with regards to metadata. The normalized frequency of structural tags is a good example- each structural tag would enjoy a fair amount of information benefit from this feature.

    a. For metadata features, some non-uniformity between structural tags is expected. Not all features can cover all entities. This is normal and expected. The goal is to maximize uniformity, not to minimize descriptive power.

5. Unique/niche sources of noise such as text should be analyzed in a descriptive manner. This follows from (4).
6. Ideally, features should be normalized in such a way that data can be easily compared between different documents. For instance, you can take the log length of text with a large log base, clamping your values at 1. This enables your models to evaluate text length between different documents in a coherent manner.
7. Use one-hot encodings only for declaring the class of the current node in the abstract syntax tree. This rule might change with time.
8. Variables should be treated with caution. All variables should be described exclusively via metadata. Ideally, metadata used to describe variables can also be used to describe other features of the code structure.
9. Finally, features should focus on the structure of the code. That is, focus on how the code looks in the abstract syntax tree. A common theme in torc is to describe the structure and usage of objects; this theme was intentionally chosen to minimize noise in what is normally a rather noisy problem.

And as with all machine learning problems, normal machine learning best practices apply. Likewise, all rules have their exceptions.

## Efficacy Testing

Due to the newness of the torc format, we needed some way of determining whether the torc format provides enough power to properly classify JavaScript files. In order to gauge the efficacy of torc, we created a large number of models with different subsets of data with different parameters to understand how models might look under different situations. The target dataset was a collection of 40,000 malicious JavaScript samples (https://github.com/HynekPetrak/JavaScript-malware-collection) and 650 “good” JavaScript

samples. The 650 good JavaScript samples were JavaScript files identified through normal browsing behavior. The model type used was a stacked LSTM feeding into a dense layer with a single sigmoid output, no dropout applied. We opted to use models with a fixed number of timesteps (100, 200, ..., 900-size timesteps). Only the number of iterations was varied between timestep categories. Note that efficacy testing was performed purely as a test and not with the intention of generating real-world models.

Example model, timestep size=400

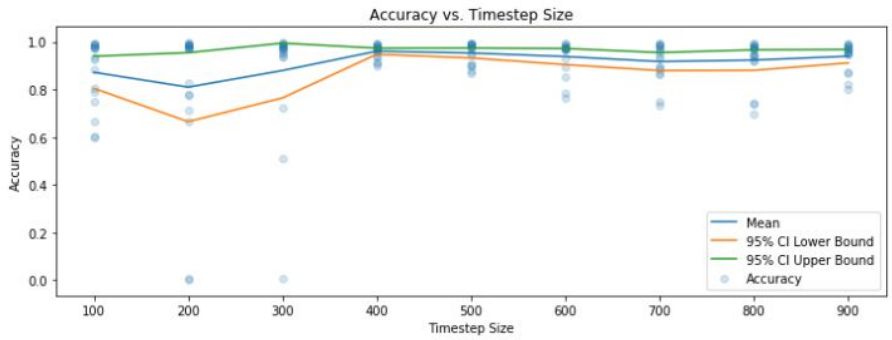
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 400, 100)          70800
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_1 (Dense)              (None, 100)               10100
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 161,401
Trainable params: 161,401
Non-trainable params: 0
```

When testing the efficacy of our format, we took our dataset and translated it to the torc format. This process took between 6 and 7 hours. We then created roughly 160 models spread over the 9 different model types previously specified (9 different timestep sizes). When training our models, we quickly ran into physical memory issues. In order to bypass these restrictions, we created a master list of 250 randomly chosen documents per class and then randomly selected roughly 60 documents from each category. We then split each document into chunks of the specified timestep size (using a rolling window), and re-sampled the minority class in order to achieve an equal number of samples. Every 15 training iterations, we selected a new batch of 60 documents from our master list and repeated the process without generating a new model. Once our model was trained, we then selected 60 random documents which were not included in the master list (via set difference to ensure no data leakage), performed the same transformations as our training data, and predicted the class for each timestep-length

subsample. Reported accuracy rates are from the test datasets. We do not provide any ROC curves as individual predictions were not stored between models.

## How Well Does It Work?

Individual accuracy points by timestep size with accuracy averages and 95% confidence intervals provided below, 17 to 18 samples per group:




| Timestep Size | Accuracy (95% CI) |
|---|---|
| 100 | 0.872 (0.805, 0.939) |
| 200 | 0.810 (0.665, 0.954) |
| 300 | 0.879 (0.764, 0.995) |
| 400 | 0.961 (0.948, 0.974) |
| 500 | 0.953 (0.932, 0.974) |
| 600 | 0.938 (0.904, 0.972) |
| 700 | 0.917 (0.879, 0.955) |
| 800 | 0.923 (0.880, 0.966) |
| 900 | 0.940 (0.911, 0.968) |

As expected, our data underperforms with fewer timesteps and begins to converge with a certain number of timesteps. With proper tuning, it should be trivial to produce a model capable of handling longer sequences in a more accurate manner.

## Current Features

Torc currently uses over 75 features. Of these features, roughly half are interesting data commonly found among different types of JavaScript files, as per our datasets. The names for this class of features are all numeric in nature. We then have roughly 20 features dedicated to certain data we explicitly flag in a given JavaScript file. These features typically represent major logic in code. For instance, we will declare when we enter a conditional statement or a try block, or when we encounter a square bracket. The remainder of our features are metadata used to describe the use of variables, the context of text strings, and other similar metadata. Of note, we include how frequently a variable is used as a function versus how frequently it is used as some data-oriented object. Among text-oriented features, we include shannon entropy and the log length of the given string. For a full list of features, please see translator.py.

## Torc Use Cases

While the proof of concept revolves around using torc as an intelligent counter-countermeasure, that is, a system designed to detect if a script is attempting to circumvent existing security controls, torc is fundamentally designed to handle a variety of use cases. In general, the use cases of torc will follow existing code analytics use cases. The key determinant in which use cases are available ultimately boils down to the features the implementer employs and whether the implementer opts to create new features. Thus, if torc is implemented with features designed to estimate the likelihood that a piece of code will break, it is theoretically possible to use torc to gauge whether a change in some code segment will result in increased errors in production. Along the same line of thought, torc can theoretically also be used to determine if a code segment is likely to cause performance problems when deployed.

## The Future of Torc

Torc has a few paths it can develop along. Of note, we can focus on horizontal expansion (onboard new languages) and vertical expansion (expand core offerings and add new features). Under the domain of new languages, torc is likely to stick to interpreted and command languages due to how tightly torc is coupled with a script's abstract syntax tree. The primary languages we would target are shell and python with shell given special priority. If we were to expand vertically, new features would focus on summarizing functions to determine function uniqueness as well as applying other interesting methods such as PageRank to score the importance of certain variables. We can also explore the ordering of the code itself.

## Contact Us

If you have any questions or are interested in getting involved, please reach out to us on github @rafields/torc. Alternatively, you can contact me by email at torc2019@gmail.com.