# TestNMT: Function-to-Test Neural Machine Translation

Robert White
Department of Computer Science
University College London
London, UK
robert.white.13@ucl.ac.uk

Jens Krinke
Department of Computer Science
University College London
London, UK
j.krinke@ucl.ac.uk

## ABSTRACT

Test generation can have a large impact on the software engineering process by decreasing the amount of time and effort required to maintain a high level of test coverage. This increases the quality of the resultant software while decreasing the associated effort. In this paper, we present TestNMT, an experimental approach to test generation using neural machine translation. TestNMT aims to learn to translate from functions to tests, allowing a developer to generate an approximate test for a given function, which can then be adapted to produce the final desired test.

We also present a preliminary quantitative and qualitative evaluation of TestNMT in both cross-project and within-project scenarios. This evaluation shows that TestNMT is potentially useful in the within-project scenario, where it achieves a maximum BLEU score of 21.2, a maximum ROUGE-L score of 38.67, and is shown to be capable of generating approximate tests that are easy to adapt to working tests.

## CCS CONCEPTS

• **Computing methodologies → Machine translation**; • **Software and its engineering → Software testing and debugging**;

## KEYWORDS

Neural machine translation, software testing

**ACM Reference Format:**
Robert White and Jens Krinke. 2018. TestNMT: Function-to-Test Neural Machine Translation. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering (NL4SE '18), November 4, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3283812.3283823

## 1 INTRODUCTION

TestNMT is an exploratory approach to test generation which aims to generate tests by learning to translate from the function domain to the test domain. However, as the model only sees the output as a sequence of tokens and has no concept of the syntax of the target programming language or test framework, there is no guarantee that the output will be a syntactically correct test. We, therefore, refer to the output as approximate tests. Given this, the goal is to allow a user to provide a function as input and receive an approximate test as output, which the developer can then manually adapt to a working test for that function.

This document provides an overview of the preliminary investigation into the potential of this approach, including the network design, data collection, and a preliminary quantitative and qualitative evaluation of TestNMT for two usage scenarios: cross-project and within-project.

The evaluation shows that, while TestNMT is most likely not useful in the cross-project scenario, it does have the potential to be of use within-project. The quantitative evaluation demonstrates this with a maximum BLEU score of 21.2, a maximum ROUGE-L score of 38.67, and edit distances between existing linked tests and generated approximate tests which show that on average approximately half of the content of the tests can be generated by TestNMT. The qualitative evaluation also shows that TestNMT has a strong potential for usefulness by demonstrating that it can produce approximate tests that are easy to adapt to the desired tests.

## 2 APPROACH

TestNMT uses the techniques of neural machine translation for natural languages and applies them to the programming language domain, specifically to translate from functions to tests. TestNMT uses an encoder-decoder with attention architecture for sequence to sequence translation, such as that shown in Figure 5 of the Tensorflow neural machine translation tutorial [3]. In this architecture the encoder encodes the source sequence into a vector representation which is then decoded into a translated target sequence by the decoder.

The encoder builds the vector representation of the source sequence by traversing the sequence one token at a time converting each token into a real-valued vector embedding via an embedding layer which is then provided as input to the encoder recurrent neural network (RNN) for that time step. After the source sequence has been fully processed, the final hidden state of the encoder RNN is used to initialise the hidden state of the decoder RNN. Then, at each time step, the decoder RNN uses the current hidden state, the previously generated target sequence token, and the attention mechanism, to generate a new target sequence token. This continues until the end-of-sequence token is generated.

To create the training data we apply a function-to-test traceability technique over a corpus of software, generating a set of example function-to-test links. These links are pre-processed into source and target token sequences which are then used to train the embedding layer and RNN units by backpropagating the sequence-to-sequence cross-entropy loss. The implementation used for these experiments was derived from the implementation provided by Luong et al. [3].

## 3 EXPERIMENTAL SETUP

The experiments are split into two scenarios: cross-project, and within-project. The cross-project scenario tests the possibility of training a single model using a large, general corpus taken from many projects, and using it to generate an approximate test for any arbitrary function. The within-project scenario tests the possibility of training a model for an individual project and using it to generate an approximate test for a function from the same project.

Experiment configuration one tests the cross-project scenario using 156 open source projects from GitHub. Configurations two, three, and four test the within-project scenario by individually using the OpenJDK 9, Netbeans, and OpenLiberty projects respectively. A natural language translation baseline for the chosen translation quality measures was obtained by also training the model for Vietnamese to English translation on a small corpus of TED talks [4].

### 3.1 Data

To train the model we need a dataset consisting of test-to-function traceability links – pairs of functions and tests where the test tests the function. As the performance of the model is dependent on the size and accuracy of the training data, we need to gather as much high-accuracy training data as possible. Ideally, the dataset would be constructed from manually labelled test-to-code traceability links but given the amount of data required this is infeasible and, therefore, we must use an automated traceability establishment technique.

When selecting an automated traceability establishment technique, we must balance precision with recall; low precision techniques will result in very noisy training data, whereas low recall techniques will result in a dataset that is too small. Both of these factors can limit the performance of the model.

*3.1.1 Generation through Naming Conventions (NC).* For this set of initial experiments, we have limited the data to Java projects with JUnit test cases and are generating the training data using variants of the Naming Convention (NC) [7] technique. NC was selected for these initial experiments as it is performed statically and should have relatively high precision on projects that use the naming convention, however, in future work a more extensive set of traceability techniques could be used to increase both the size and precision of the dataset.

The two variants of NC that we tested are Strict NC and Relaxed NC. Strict NC matches both the class names and the function names, for example, the *toString()* function in the class *Point* will only be matched to a test called *testToString()* or *toStringTest()* in a test class called *TestPoint* or *PointTest*. Relaxed NC however, matches only on the function name, so *toString()* in any class will match to *testToString()* or *toStringTest()* in any test class. Therefore, Strict NC is better for precision, while Relaxed NC is better for recall.

*NC Generation Experiments.* Table 1 shows the results of an experiment comparing the sizes of the datasets generated by Strict NC and Relaxed NC on a set of popular open source Java projects.

We can see from these results that while projects which closely follow the naming convention, such as JFreeChart and Commons Lang, may produce a good amount of links for Strict NC, most projects do not follow the convention closely and produce little to no links for Strict NC. Overall the number of links found using Strict NC is not enough for training a model.

**Table 1: Number of test-to-code trace links generated by naming convention techniques.**

| Project | JUnit Tests | Strict NC | Relaxed NC |
|---|---|---|---|
| Apache Poi | 1,582 | 0 | 10,546 |
| JFreeChart | 2,482 | 1,016 | 1,016 |
| Closure Compiler | 153 | 56 | 101 |
| Commons Lang | 3,061 | 2,385 | 10,647 |
| Commons Math | 4,461 | 770 | 10,033 |
| Commons Collections | 2,661 | 455 | 12,167 |
| Eclipse CDT | 856 | 42 | 1,759 |
| Android Platform | 4,021 | 1,128 | 34,711 |
| Chromium | 6,334 | 435 | 7,347 |
| Netbeans | 1,582 | 399 | 57,156 |
| Total | 28,500 | 5,688 | 145,828 |

When we switch to Relaxed NC the number of links increases to levels that make training a model feasible. While this increase in links has a concomitant increase in noise due to false positives (functions and tests that are incorrectly matched), some of the false positives may provide useful information for the model in cases where the test is related to the function even though it does not directly test it. One example of this is typically overridden methods (toString, hashCode, equals), which should all share a similar structure and relationship to their tests. Therefore these links can provide useful information for the network to learn even if they are false positives. Given this, we selected Relaxed NC to generate the data for our experiments.

*3.1.2 Preparation.* The data for the experiments was obtained by first applying the Relaxed NC technique over the subject(s) to establish the traceability links. The source of the functions and tests was then pre-processed to remove camel-casing, convert all strings to lower case, remove all numbers, and add spaces between all words and symbols. Programming language keywords and syntax were retained. Camel-cased tokens were split into individual tokens in an effort to keep the vocabulary sizes manageable as the individual tokens are more likely to appear in multiple sequences than the complete camel-cased tokens, the same applies to adding spaces between words and symbols. Numbers were removed as some exploratory experiments showed that their inclusion degraded the results. The pre-processed sequence pairs are then split between the training and test sets and all training set pairs that contain a source or target sequence that also appears in the test set are removed, this helps to avoid overfitting.

### 3.2 Network Configuration

The network configuration used for the experiments set the dimensions of the embeddings at 128, the number of RNN layers at 2, dropout at 0.2, and uses an attention mechanism named Scaled Luong, a scaled variant of the attention mechanism described in [5]. This network configuration was selected simply as a default reference; the optimisation of the network configuration has not yet been explored. The maximum sequence length for training was set to 50 tokens for the baseline natural language translation and 200 tokens for the test translation experiments, with any longer sequences are truncated to this limit. This difference is due to the fact that the functions and tests are significantly longer than the natural language sentences. This sequence length limitation hinders the learning of any relationships that are predominantly found in long sequences, after the 200 token limit. Therefore, increasing the maximum sequence length could improve the results, however, it also increases the training time which made using a sequence length longer than 200 infeasible for these initial experiments.

## 4 EVALUATION

We evaluate the performance of TestNMT both quantitatively and qualitatively. The quantitative evaluation is carried out using the standard techniques for measuring the quality of translations BLEU [6] and ROUGE-L [2]. BLEU is a precision-based measure which utilises a modified n-gram precision to calculate the co-occurrence of tokens in the candidate and reference sequences. Clipping is applied at the frequency of the n-gram in the reference sequence. ROUGE-L uses the Longest Common Subsequence (LCS) between the candidate and reference to calculate the precision and recall of the matching unigrams which are then used to calculate the F-Score. At each training step, these measures are calculated over all instances in the test set and averaged to get the scores for that step. The maximum scores for each experiment are reported in the results.

The quantitative evaluation also includes statistics for the average length of the pre-processed linked tests and the median edit distance from the generated approximate tests to these linked tests for the functions from the test set. This gives us a metric for how close the approximate tests generated by TestNMT are to the linked tests, therefore indicating the potential usefulness of TestNMT in a real-world scenario.

The qualitative evaluation is conducted by inspecting some example generated approximate tests and comparing them to the pre-processed linked tests for the input function to determine the type of edits that the developer would have to make to adapt the approximate tests for use.

### 4.1 Quantitative Results

The results from the quantitative evaluation are presented in Table 2. The Config 1 results indicate that the cross-project scenario may not be feasible as the maximum BLEU (1.3) and ROUGE-L (16.3) scores are very poor and the median edit distance is larger than the average length of the pre-processed linked tests. These numbers also saw no improvement during training showing that no significant translation is occurring and suggests that no amount of training will improve the scores at this dataset size of ~750k training examples. It is possible that more training data will bring this scenario into a reasonable range of performance but the dataset may have to be extremely large.

Configs 2, 3, and 4 indicate that TestNMT has significant potential for usefulness in the within-project scenario as the maximum BLEU scores are equivalent to that of the Vietnamese to English natural language translation baseline, and the ROUGE-L scores average 73% of the baseline score, which is also a strong result. The edit distance results also reinforce this conclusion by indicating that, on average, the edit distance between the generated approximate tests and the pre-processed linked tests is within ~50% of the length of the pre-processed linked tests. This demonstrates that TestNMT is producing useful content and indicates that using it should save the developer time over creating tests from scratch.

### 4.2 Qualitative Results

Here we present some examples taken from the approximate tests generated by the trained models in each experiment. The "linked test" is the test from an example link drawn from the test set after the data pre-processing has been applied. The "approximate test" is the output of the model when provided with the function from the example link as input. The differences are highlighted with red to indicate deletions and green to indicate insertions in the diffs between the linked and generated approximate tests.

Examples from Config 1 are not included as the model did not train, however, configs 2 and 4 demonstrate the potential of TestNMT in the within-project scenario as we can see that there are only a few differences between the linked and approximate tests. For the first example of Config 2, the differences are only a change in the naming of the test case and the arguments for one function call. For the second example of Config 2, there is just a type change to a related exception type and a change to an assert condition. The only differences in the examples from Config 4 are two string literals and two changes to an assert.

#### 4.2.1 *Config 2 – Example 1.* Pre-processed linked test:

```
@ test public void test _ minus _ long period unit _ zero () { local
date time t = test _ _ _ _ _ _ _ . minus ( ,chrono unit . days ) ;
assert same ( t , test _ _ _ _ _ _ _) ; }
```

Generated approximate test:

```
@ test public void test _ minus _ period _ zero ( ) { local date
time t = test _ _ _ _ _ _ _ . minus ( mock simple period . zero _
days ) ; assert same ( t , test _ _ _ _ _ _ _ ) ; }
```

#### 4.2.2 *Config 2 – Example 2.* Pre-processed linked test:

```
@ test public void test ( ) { sql transaction rollback exception ex
= new sql transaction rollback exception ( ) ; assert true ( ex . get
message ( ) = = null & & ex . get sql state ( ) = = null & & ex .
get cause ( ) = = null & & ex . get error code ( ) = = ) ; }
```

Generated approximate test:

```
@ test public void test ( ) { sql data exception ex = new sql data
exception ( reason ) ; assert true ( ex . get message ( ) . equals (
reason ) & & ex . get sql state ( ) = = null & & ex . get cause ( )
= = null & & ex . get error code ( ) = = ) ; }
```

#### 4.2.3 *Config 4 – Example 1.* Pre-processed linked test:

```
@ test public void get unique group id ( ) throws exception { string
group = " vmmgroup " ; string unique group id = "cn = vmmgroup ,
ou = users , dc = rtp , dc = raleigh , dc = ibm , dc = com " ; log . info (
 c , " get unique group id " , " checking with a valid group . " ) ;
 assert d ns equal ( null , unique group id , servlet . get unique
group id ( group ) ) ; }
```

Generated approximate test:

```
@ test public void get unique group id ( ) throws exception { string
group = " vmmgrp " ; string unique group id = "cn = vmmgrp , o = ibm ,
c = us " ; log . info ( c , " get unique group id " , " checking with
```

**Table 2: TestNMT experimental results.**

| | Baseline (VI-to-EN) | Config 1 | Config 2 | Config 3 | Config 4 |
|---|---|---|---|---|---|
| | Data | | | | |
| Project(s) | N/A | 156 OSS | OpenJDK 9 | Netbeans | OpenLiberty |
| Train Set Size | 133k | 744k | 105k | 11k | 88k |
| Test Set Size | 1500 | 300 | 100 | 100 | 100 |
| Source (Func.) Vocab Size | 17k | 24k | 9k | 6k | 7k |
| Target (Test) Vocab Size | 8k | 13k | 2k | 2k | 4k |
| | Results | | | | |
| Max BLEU | 20.24 | 1.3 | 19.51 | 21.20 | 19.60 |
| Max ROUGE-L | 50.40 | 16.3 | 38.3 | 33.73 | 38.67 |
| Avg. Linked Test Len. | N/A | 375 | 522 | 543 | 625 |
| Median Edit Dist. | N/A | 1431 | 246 | 344 | 269 |

```
 a valid group . " ) ; assert d ns equal ( unique group id , servlet
  . get unique group id ( group ) ) ; }
```

*4.2.4 Config 4 – Example 2.* Pre-processed linked test:

```
@ test public void get groups _ empty list ( ) throws exception {
search result result = reg . get groups ( " * " , ) ; assert not null
 ( " search result must never be null " , result ) ; assert true (
result . get list ( ) . is empty ( ) ) ; assert false ( result .
 has more ( ) ) ; }
```

Generated approximate test:

```
@ test public void get groups _ empty list ( ) throws exception {
search result result = reg . get groups ( " * " , ) ; assert not
null ( " search result must never be null " , result ) ; assert
equals ( , result . get list ( ) . size ( ) ) ; assert false ( result
 . has more ( ) ) ; }
```

## 4.3 Results Discussion

Overall these preliminary results are encouraging and show that although TestNMT may not be able to produce complete and finished tests that can be plugged directly into the system in question, it has the potential to produce recommendations that are very close to the required test. A developer may then only have to make small changes to the generated approximate test in order to adapt it for use in the system.

## 5 FUTURE DIRECTIONS

There are a few key areas in which this research can be expanded to produce an extensive and robust investigation. Firstly no attempt has yet been made to optimise the network architecture or hyperparameters, which could have a large impact on the results. Aspects to explore in this area include using different types of RNN units, such as GRU, DGRU, or peephole LSTM units, as well as greater numbers of layers and different attention mechanisms. Bi-directional units should also be tested as previous works have noted the effect that the order of the input sequence can have [8], and the effectiveness of Bi-directional units in exploiting this [1]. The use of Beam Search could also be investigated.

The evaluation also needs to be extended to confirm the findings of the cross-project scenario with a larger dataset and confirm the findings of the within-project scenario with a wider range of projects of varying sizes. This will facilitate the application of statistical analysis to strengthen the results and empirically test the effect of project size on the results. This analysis could also provide insights into any other properties of a project that may affect its viability for use with TestNMT and more generally uncover any currently hidden problems with the approach.

Further, as TestNMT learns from human-written tests that contain elements that are difficult or impossible for typical test generation tools to produce, such as oracles and specific test inputs, TestNMT may be more useful than typical test generation tools for generating these types of elements. Testing this hypothesis should also be addressed in future work.

## 6 CONCLUSION

This preliminary investigation into the performance of TestNMT and, more generally, the viability of using neural machine translation as a test generation technique has shown that it has the potential to be of use, especially when applied to large individual projects. However, there is still much work to be done to optimise the implementation and carry out a full evaluation to determine the overall benefit and generality achievable by TestNMT.

## REFERENCES

[1] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural Machine Translation by Jointly Learning to Align and Translate. (2014), 1–15. https://doi.org/10.1146/annurev.neuro.26.041002.131047 arXiv:1409.0473
[2] C Y Lin. 2004. Rouge: A package for automatic evaluation of summaries. *Proceedings of the workshop on text summarization branches out (WAS 2004)* 1 (2004), 25–26.
[3] Minh-Thang Luong, Eugene Brevdo, and Rui Zhao. 2017. Neural Machine Translation (seq2seq) Tutorial. *https://github.com/tensorflow/nmt* (2017).
[4] Minh-Thang Luong and Christopher D. Manning. 2015. Stanford Neural Machine Translation Systems for Spoken Language Domain. In *International Workshop on Spoken Language Translation.* Da Nang, Vietnam.
[5] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2015. Effective Approaches to Attention-based Neural Machine Translation. (2015). https://doi.org/10.18653/v1/D15-1166 arXiv:1508.04025
[6] Kishore Papineni, Salim Roukos, Todd Ward, and Wj Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. *ACL '02 Proceedings of the 40th Annual Meeting on Association for Computational Linguistics* July (2002), 311–318. https://doi.org/10.3115/1073083.1073135 arXiv:1702.00764
[7] Bart Van Rompaey and Serge Demeyer. 2009. Establishing traceability links between unit test cases and units under test. *Proceedings of the European Conference on Software Maintenance and Reengineering, CSMR* (2009), 209–218. https://doi.org/10.1109/CSMR.2009.39
[8] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems (NIPS)* (2014), 3104–3112. https://doi.org/10.1007/s10107-014-0839-0 arXiv:1409.3215