Accredited Ranking SINTA 2
Decree of the Director General of Higher Education, Research, and Technology, No. 158/E/KPT/2021
Validity period from Volume 5 Number 2 of 2021 to Volume 10 Number 1 of 2026

Published online on: **http://jurnal.iaii.or.id**

JURNAL RESTI
(Rekayasa Sistem dan Teknologi Informasi)
Vol. 6 No. 2 (2022) 305 - 314 ISSN Media Electronic: 2580-0760

Towards Generating Unit Test Codes Using Generative Adversarial Networks

Muhammad Johan Alibasa[1], Rizka Widyarini Purwanto[2], Yudi Priyadi[3], Rosa Reska Riskiana[4]
[1,3,4]School of Computing, Telkom University
[2]School of Computer Science and Engineering, The University of New South Wales
[1]alibasa@telkomuniversity.ac.id, [2]r.purwanto@unsw.edu.au, [3]whyphi@telkomuniversity.ac.id,
[4]rosareskaa@telkomuniversity.ac.id

Abstract

Unit testing is one of the important software development steps to ensure the software's quality. Despite its importance, unit testing is often neglected since it requires a significant amount of time and effort from the software developers to write them. Existing automated testing generating systems from past research still have shortcomings due to the Genetic Algorithm (GA) limitations to generate the appropriate unit test codes. This study explores the feasibility of using Generative Adversarial Networks (GAN) models to generate unit test code with the ability of GAN to cover GA's drawbacks. We perform experimentations using four state-of-the-art GAN models to generate basic unit test codes and compare the results by analyzing the generated output codes using novel metrics proposed from past studies as well as performing qualitative evaluation on the generated outputs. The results show that the generated codes have satisfactory quality scores (BLEU-2 of around 99%) from the models and adequate diversity score (NLL-Div and NLL-Gen) in most models. Our study shows positive indications and potential in the use of GAN for automatic unit test code generation and suggests recommendations for future studies in GAN-based unit test code generation systems.

Keywords: unit test, code generation, generative adversarial network

1. Introduction

Unit testing is one of important processes in software development because it is a preventive measure to find issues early, which will be easier to resolve than when all units have been integrated. Still, unit testing is not always run by all software developers (programmers). According to a survey paper [1], some companies are still reluctant to require programmers to do unit testing because the benefits of applying unit testing cannot be assessed quantitatively based on the calculation of Return on Investment (RoI). The results of the qualitative analysis [2] also show that writing code for unit testing requires a learning curve and experience so that beginners are hesitant and tend to avoid unit testing. Another study [3] also showed that novice developers showed negative affective reactions when they are required to always implement unit testing. Therefore, companies and programmers still doubt whether the effort and time allocated to write unit test code is worth the added value generated [1].

To minimize the effort and time required for unit testing, research has been done to build systems that generates unit test code automatically. One example is the EVOSUITE system [4] which was developed to synthesize unit test code for the Java programming language (JUnit). Before this system was developed, previous systems only focused on one coverage target at a time, which led to a lot of redundancy in unit test code and low maintainability. To find the most optimal coverage criteria, EVOSUITE system uses a search-based approach with Genetic Algorithm (GA). The next study [5] developed a better system by adding the Many-Objective Sorting Algorithm (MOSA) to the GA so that the system can search for target coverage that has not been covered more optimally. Another system [6] was also developed with a similar approach to EVOSUITE but focuses on synthesizing unit test code for the Python programming language. Similar to EVOSUITE, this system also uses GA to get unit test code results that produce the best test coverage or code coverage.

Accepted: 14-03-2022 | Received in revised: 16-04-2022 | Published: 29-04-2022

The approaches used in the previous study has some shortcomings and challenges. First, there are challenges regarding execution time. Previous studies used GA that requires relatively long computational time and cannot always produce optimal solutions [7]. In addition, a study conducted by Almasi et al. [8] shows that the unit test code generated by EVOSUITE can only detect problems up to 56.4% in product codes from their case study. According to the survey results, the EVOSUITE has not yet reached a standard sufficient for industrial use, even though the system is heading in the right direction. The study conducted also found that the system still has problems for complex program code.

Generative Adversarial Networks (GAN) is one of the machine learning algorithms that has started to become the main focus for synthesizing new data [9]. In recent years, GAN has often been used to generate images [10] and synthetic videos [11], [12]. In addition, according to a literature study [13], GAN can also be used to produce a text. The text can be a free sentence that has meaning or a description of an image and video. This motivated our research to explore the feasibility of GAN in automating unit test code generation. The use of GAN can solve the drawbacks of GA in some cases. For example, GAN takes a long time to "learn" the process, but the output search process will be much faster than a GA-based approach.

Our paper makes three contributions: (1) four state-of-the-art GAN models that generates simple unit test codes, (2) thorough analysis of result comparisons from the models, and (3) recommendations for future studies. The first contribution is a novel attempt to explore whether GAN models are able to generate simple assertion method codes. It is important to check the feasibility from a simple task first then expand the scope into harder tasks. This reason leads this study to focus more on simple code generation tasks. For the second contribution, this paper provides analysis on three novel metrics from the previous studies that can be used in text generation problems (Section 3.4.1 to 3.4.3), and the paper also includes manual qualitative judgements to check if the generated codes contain any error (e.g., syntax error). For the last contribution, the paper brings discussions for future studies in this direction based on the results found in this study. The recommendations provided are well-founded as they are established from our experiments and new results.

Related Work

Generative Adversarial Networks (GAN) is a trending model for semi-supervised and unsupervised learning processes [14]. In general, GAN consists of two components that compete with each other, namely generator and discriminator. The generator component focuses on generating realistic data, such as an image. Meanwhile, the discriminator component focuses on distinguishing between the real data and the fabricated data generated from the generator component [9]. GAN is commonly used to generate new images or videos. For instance, StoryGAN [10] was developed to generate a sequence of images corresponding to the text or story inputs. The model in this study included a deep context encoder to the conditional architecture of GAN (Mirza & Osindero, 2014), to help understand the input in the form of a story text. In addition to the previous type of encoder, Variational Autoencoder (VAE) can also be added for the video synthesis process from text input [12]. Another study [11] added a text-filter process to the conditional GAN so that it could produce a better video synthesis from text than previous studies.

According to a literature study [13], GAN has been applied several times for the text generation process. In general, the research that has been done uses three approaches, namely Gumbel-Softmax differentiation, Reinforcement Learning, and modified training objectives. The main challenge in text synthesis is the intrinsic features of a language, such as its grammar, syntax, and semantic properties. One of the frameworks [15] for text synthesis was developed by utilizing the conditional structure of GAN. The framework accepts an image as input to generate text that describes the input image. Using this model, text that is more diverse and natural can be produced so that it looks more like human expressions.

Several other studies also attempted to make the text produced better using GAN. DPGAN or Diversity-Promoting Generative Adversarial Network [16] can produce more diverse, new, and informative texts than studies in the previous year. The dataset used comes from comments on Yelp, Amazon, and OpenSubtitles websites. Another study [17] utilized a generator based on relational memory and Gumbel-Softmax relaxation. Such studies can produce texts that are better at following correct grammar and have clear meanings.

In addition to the general text generation, GAN has also been used for the synthesis process of a code or source code. Liu et al., [18] developed TreeGAN which can produce text with sequences that follow grammatical rules. The TreeGAN utilizes the Recurrent Neural Network (RNN) in the generator section and tree structured RNN in the discriminator section. The study was able to generate random SQL queries and Python language code with fewer syntax errors when compared to the SeqGAN algorithm [19]. The results of the quantitative and qualitative analysis of the study indicate that TreeGAN has a slightly better performance than SeqGAN.

2. Research Methods

As shown in Figure 1, our methods can be divided into five major parts, that consist of designing model's architecture, generating dataset (training and testing),

DOI: https://doi.org/10.29207/resti.v6i2.3940

pre-processing the data, training the models and evaluating the performance of each model built.

2.1 Selected GAN Models

This study builds four GAN models, that includes MaliGAN, SeqGAN, DPGAN, and JSDGAN. The following subsections describe each model separately in more detail.

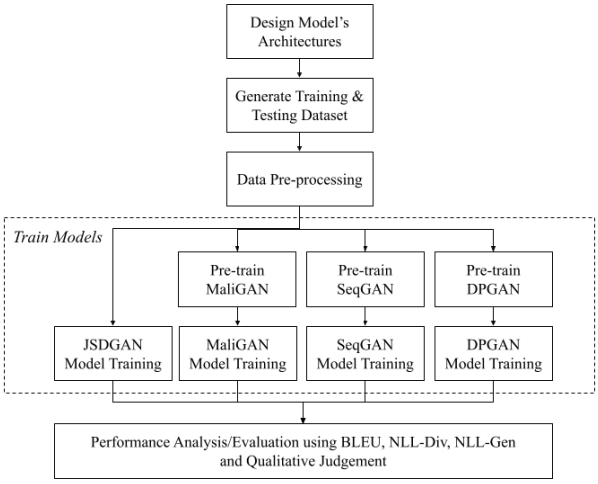

Figure 1. Research Model

2.1.1 MaliGAN

Che et al. [20] modified the training objective of the generator part of GAN to optimize using a different objective, that is using importance sampling thus it helps the training procedure to be closer to maximum likelihood (MLE) training of auto-regressive models. By using this approach, the model is more stable and has less variance in the gradients compared to directly optimizing the standard GAN objectives. The idea of this importance sampling procedure was inspired from another study by Hjelm et al. [21].

Figure 2 shows the structure of MaliGAN model. The generator part of the model is similar to the standard GAN model, but the discriminator has a novel gradient estimator that is shown in the equation inside the figure. Based on the study result [20], the model produces positive results when it is used on sentence-level language modelling. The model performed more stable during training and could achieve better score in terms of perplexity.

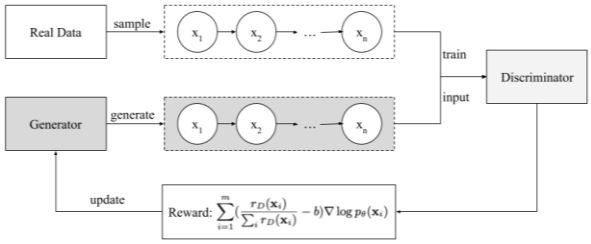

Figure 2. MaliGAN Architecture

2.1.2 SeqGAN

Yu et al. [19] proposed SeqGAN model specifically designed to generate sequence such as texts of Chinese poems and Barack Obama political speeches. In their study, they used recurrent neural networks (RNN) as generative model and leverage the Long Short-Term Memory (LSTM) to implement the update function. Meanwhile, they used CNN for the discriminator as it has been shown that CNN has great effectiveness in text classification tasks [22].

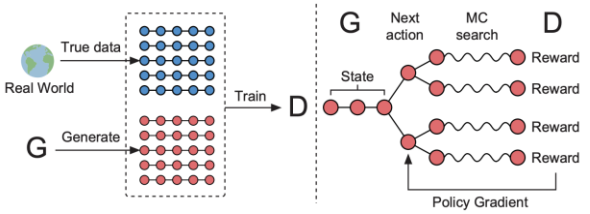

Figure 3. SeqGAN Architecture [19]

Figure 3 shows the illustration on how SeqGAN differs compared to the standard GAN architecture. The discriminative part received both real data sequences and negative samples from the generator part during the training process. Concurrently, the generator part is being updated by using a policy gradient and Monte Carlo search based on the reward value obtained from the discriminator part.

2.1.3 DPGAN

Diversity-Promoting Generative Adversarial Network (DPGAN) was proposed by [16] to generate "novel" and fluent text. The model penalized repeated generated texts by applying low reward and encouraged diverse and informative texts by applying high reward. The generator part of this model used a standard LSTM decoder. In contrast, the discriminator part utilized a unidirectional LSTM, a language-based discriminator. During the training, the model maximizes the reward of real-world texts and minimizes the reward of generated texts. This approach will prevent the model to generate novel texts with low quality.

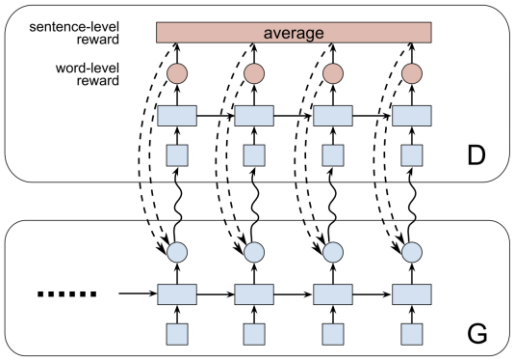

Figure 4. DPGAN Illustration [16]

DPGAN illustration (Figure 4) shows that the reward function consists of two parts: sentence-level reward

DOI: https://doi.org/10.29207/resti.v6i2.3940

and word-level reward. One of the major common issues in the standard discriminator part is that the reward for high-novelty text is easy to saturate leading to problems detecting novel texts. The proposed discriminator in this model showed better performance in distinguishing novel texts without having the saturation problem. To produce better results, the word-level reward part generates different reward for any different words in a particular sentence.

2.1.3 JSDGAN

Jensen-Shannon Divergence (JSD) GAN is proposed by Li et al. [23] and has a unique trait compared to other GAN architecture. The model excludes the explicit neural network for the discriminator part. Instead, the model used an alternative mini-max optimization procedure for the distinguishable game value function so that the maximization step includes a closed form solution for the discriminator part. This process is equal to directly optimizing the Jensen-Shannon divergence (JSD) between the generator's distribution and the real-data distribution from the training data without the generator sampling. This model is found to have better performances compared to other discrete sequence generation models.

As shown in Figure 5, the JSDGAN architecture does not include any explicit discriminator part. The reward was calculated using the equation shown in the figure that compute the gradient of JSD. The equation is a modification of the gradient the log-likelihood. The model used stochastic gradient descent (SGD) to optimize the JSD between the distribution of generator output and the empirical training data.

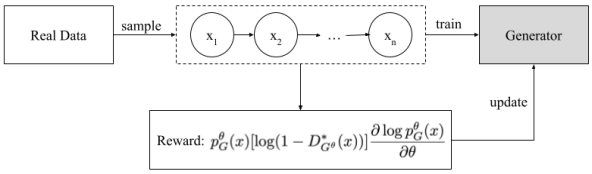

Figure 5. JSDGAN Architecture

2.2 Dataset

This study aims to explore the feasibility of using GAN to generate an actual unit test code. In the most basic form, unit test code consists of assert method calls. The assert method checks whether the output of a function or class method produces the expected results. This study runs experiments to observe whether the models in the previous subsection are able to generate simple assert method calls without any syntax error. Past study [18] found that the method produces code with fewer syntax errors, but the GAN model was trained with random lines of Python code. More importantly, not all generated codes are free from syntax errors due to the codes used for the training process are diverse.

As there is currently no dataset available that consists of python assert method calls, we generated our own dataset that consists of 20,000 lines for training set and 10,000 lines for testing dataset, and each line is a Python assert method call (unittest module). The arguments of each assert method were randomized with many variations of argument types, including arbitrary value (number, string or boolean), variable names, function call, object property and object method call. The variable and method names were randomized from a list common name for variables or methods, e.g., [24]. Some examples of these python assert method calls in the dataset are shown in Table 1. Both training and testing dataset are available online in our organization research Dataverse[1] for future studies to use and reproduce.

Table 1. Training Dataset Sample

Assert Method Call
assertNotIn(find_result(False, True, position8), list3)
assertIn(call0.retrieve_input(error4, 'up'), length2)
assertIsNone(x)
assertNotEqual(send_result(arr7, True, position5), 158)
assertTrue(i.is_complete(True))
assertFalse(input2)
assertEqual(size, point8)
assertTrue(test9.send_output(i))
assertNotEqual(var4, 671)
assertFalse(add_input(True, True))

2.3 Experiment

We tested four different models as specified in Section 3.1 in this paper. Before feeding the dataset into the model, the dataset was pre-processed to add a single white space for each word or special character related to the Python syntax, e.g., '(', ')', ',', '.', and others (excluding underscores). The white space is required so that the tokenization process will separate these characters with the method and variable names. Each word or special character is then converted into a token by using tokenization method from Natural Language Toolkit (NLTK). For each tokenization, we keep both mapping from word to token index and vice versa. Afterwards, they were ready to be used as input to the GAN models.

The GAN models were built using PyTorch library and were based on each past paper provided code. We hyper parameterized these four models and used parameter values that are shown in Table 2. There are two stages of training process to build the models. The first stage is pre-training both the generator and the discriminator that helps GAN to train much better. During this pre-training, the discriminator was trained to minimize its cross-entropy while the generator was trained to perform Maximum Likelihood Estimation (MLE). Only

[1] *link was removed due to anonymization*

DOI: https://doi.org/10.29207/resti.v6i2.3940

JSDGAN does not include this pre-training stage. The second training stage is the adversarial training where the discriminator competes against the generator to approach Nash equilibrium. The second stage is the standard GAN process that consists of two parts: the discriminator was trained to classify train data vs. generated data, and the generator was trained to capture essential patterns of the training set. From this adversarial training, the models generated samples that are similar to real data distributions. The whole training process was limited to 27.5 hours since we also want to compare the number of epochs generated during the same amount of time from all models.

Table 2. Hyperparameter for GAN Models

Parameter	Value
Generator (GEN) Init	'normal'
GEN learning rate	0.01
GEN embed dimension	32
GEN hidden dimension	32
Discriminator (DIS) Init	'uniform'
DIS learning rate	0.01
DIS embed dimension	64
DIS hidden dimension	63
Max sequence length	20
Batch size	8

2.4 Evaluation Metrics

We use three metrics to evaluate and analyze each model's results. The metrics used are BLEU, NLL-Gen and NLL-Div. In addition to these metrics, we also evaluate the generated samples based on qualitative judgement, as indicated in another study [25] that text generation problems require human perspective to evaluate the generated output manually.

2.4.1 BLEU

The first metric is BLEU or Bilingual Evaluation Understudy Score [26], a metric that can be used to evaluate the generated texts from GAN compared to a reference sample or sentence [27]. The score ranges between 0 to 1, where 1 indicates a perfect match compared to human reference samples while 0 means a perfect mismatch. BLEU is often used in GAN model evaluation since it is fast, language independent, and easy to understand. BLEU also often correlates highly against human manual evaluation. This metric is widely used and NLTK library provides an implementation of the BLEU score making it easier to use. In this paper, we use cumulative BLEU scores, from BLEU-2 to BLEU-5. The cumulative score is obtained from the calculation of individual n-gram scores from 1 to N and weighted by calculating the weighted geometric mean. For instance, BLEU-2 score assigns weight 50% to each 1-gram and 2-gram scores, and BLEU-5 assigns 20% to each 1-gram, 2-gram, 3-gram, 4-gram and 5-gram scores. To obtain the individual n-gram score, we need to evaluate the matching gram of a particular order, for example, 1-gram (a single word) or 2-gram (word pairs). Equation (1) and (2) show how to compute BLEU score,

$$BLEU = BP \cdot \exp\left(\sum_{n=1}^{N} w_n \log p_n\right) \quad (1)$$

$$BP = \begin{cases} 1 & if\ c > r \\ e^{(1-\frac{r}{c})} & if\ c \leq r \end{cases} \quad (2)$$

where p_n is the calculated geometric average of the modified n-gram precision (up to length N), w_n is the assigned weight as shown in the previous example, c is the length of the compared text or the generated sample, and r is the length of reference text or real-data sample.

Since the text generation in our study is unconditional, all lines of code in the test set are used as the reference for the BLEU calculation. One issue from using BLEU in unconditional text generation is that the BLEU score only considers the validity of generated texts without measuring the proportion of the reference texts that can be covered by the models. As GAN may generate texts that are similar or from a limited set of texts, we need to use other metrics to evaluate the diversity of the outputs generated from the models.

2.4.2 NLL-Div and NLL-Gen

NLL-Oracle was introduced from the study that proposed SeqGAN [19]. This metric evaluates and considers a random distribution as the real distribution (oracle) and the training set is used by sampling from this distribution. The score was calculated using NLL or Negative Log Likelihood from the generated samples obtained from the trained model. However, the metric is not considering the coverage or the diversity, thus metrics named NLL-Div and NLL-Gen were proposed.

NLL-Div can be used to evaluate the diversity of the generated samples [25, 28]. The metric also calculates the negative log likelihood of the generated samples using equation (3),

$$NLL_{Div} = -\mathbb{E}_{Y_\theta \sim P_\theta}[\log P_\theta(y_1, \ldots, y_T)] \quad (3)$$

where y_T is the sample from the real data distribution and P_θ is the generated sample distribution. NLL-Div is able to check whether the generated samples contain repeated texts, thus will be able to evaluate the diversity of the generated outputs. The low value of this metric indicates that the generated samples were obtained from a limited set of patterns from the real data set, or the generator assigns all its probability mass to a small region.

NLL-Gen [29] is also used in this study to evaluate the diversity of the generated samples. NLL-Gen is the reverse direction of NLL-Oracle, so it is sensitive to the diversity and not the quality. To evaluate the quality, we have included BLEU metric that described previously in this section. The NLL-Gen is defined as shown in equation (4),

DOI: https://doi.org/10.29207/resti.v6i2.3940

$$NLL_{Gen} = -\mathbb{E}_{Y_r \sim P_r}[\log P_\theta(r_1, \dots, r_T)] \quad (4)$$

where P_θ is the generated data distribution and r_T is the sample from the generated data. Since NLL-Gen is in the reverse direction, the lower score from this metric means a better performance, while the higher score means a worse performance.

3.4.3 Qualitative Judgement

In addition to the previous metrics, we also use qualitative judgement where we evaluate the results or generated code qualitatively based on particular aspect. GAN models are trained to produce realistic texts thus the models do not optimize for traditional cross-entropy loss. By using our judgements, we can evaluate if the generated samples from the selected models are realistic and error free. From this approach, we also analyze the generated samples to examine which model that produces more complex assert method calls. More importantly, we also want to see if the generated code has any syntax error or problems.

3. Results and Discussion

3.1 Quantitative Results

Table 3 shows the metric results from all four models. The results shown were from the iteration or epoch when the generator loss was the lowest across all iterations. Based on the table, almost all models produced BLEU-2 score close to 1 or near perfect score. This result is reasonable since the word pairs generated from all models were very similar to the reference code from the test set. The performances for BLEU-3 across all models were still positive as they are higher than 95% and the differences were less than 3%. However, the results dropped by observing BLEU-4 and BLEU-5 scores from three models where they declined about 10% and 20% from the previous cumulative score, respectively. The JSDGAN model was able to keep performance above 90% until 4-gram before dropping about 20% in 5-gram BLEU score. The tables also shows that the JSDGAN model performed better compared to other models considering the cumulative score 3-gram to 5-gram. This indicates that the codes produced from this model were in higher quality or more similar to the testing dataset.

Table 3. The Performance Comparison Across All Models

Metric	MaliGAN	SeqGAN	DPGAN	JSDGAN
BLEU-2	0.997	0.993	0.993	0.995
BLEU-3	0.956	0.956	0.957	0.976
BLEU-4	0.850	0.864	0.850	0.932
BLEU-5	0.646	0.680	0.684	0.716
NLL-Div	0.711	0.655	0.760	0.333
NLL-Gen	0.773	0.794	0.759	3.633

While the results from the JSDGAN model were great in terms of quality, the results from the diversity aspect were poor from this model. The model had an NLL-Div score of 0.333 that is about half of the results from other models. This score indicates that the codes generated from JSDGAN are not diverse and they are from a limited set of patterns. This issue is also observed when we qualitatively analyze the generated code from this model in the next subsection. The diversity issue is also found from the NLL-Gen score (3.633) as higher value indicating worse performance in the diversity aspect. The other three models had similar performance in terms of NLL-Div score, but DPGAN had the best score with 0.760 (the highest). Similarly, the DPGAN model showed the best NLL-Gen score showing the lowest score of 0.759. However, the difference is less than 0.05 so the three models (MaliGAN, SeqGAN and DPGAN) have similar performance regarding the generated code diversity.

By analyzing the BLEU results of MaliGAN model as shown in Figure 6, the scores were quite stable without any significant drop across all iterations or epochs. There are two noticeable fluctuations from BLEU-4 and BLEU-4. The BLEU-4, in particular, had oscillation with a range of 8% (between 63% to 71%), but it is fair to conclude that the quality did not decrease much across all iterations.

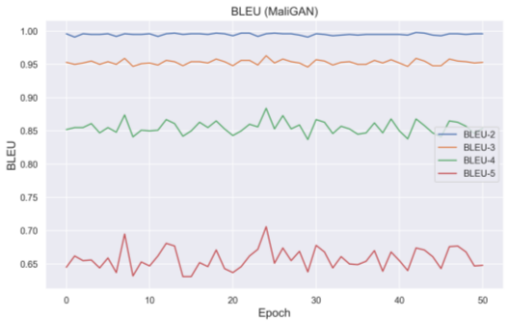

Figure 6. MaliGAN BLEU Scores

Similarly, the BLEU scores from SeqGAN model were also quite stable (Figure 7). The BLEU score variations are also relatively identical compared to MaliGAN model. The main difference between these models is that SeqGAN model had a rising trends if we observe the moving average of BLEU-4 and BLEU-5 scores.

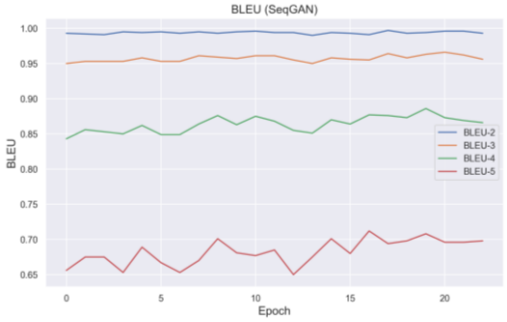

Figure 7. SeqGAN BLEU Scores

As shown in Figure 8, DPGAN model also displayed a positive trend for both BLEU-4 and BLEU-5 scores. Particularly BLEU-5, the score varies from lower than

DOI: https://doi.org/10.29207/resti.v6i2.3940

70% in the early iterations to higher than 75% in the latter iterations. The results indicate that the results of DPGAN might increase if the number of iterations is higher. The results from JSDGAN (Figure 9), however, were more interesting. By observing the BLEU-5 score, it showed a declining trend as the number of iterations increased. The negative trend is more obvious when we analyze the moving average. This finding leads us to observe the codes generated in the latter iterations. The discussion regarding this observation is included in the next subsection.

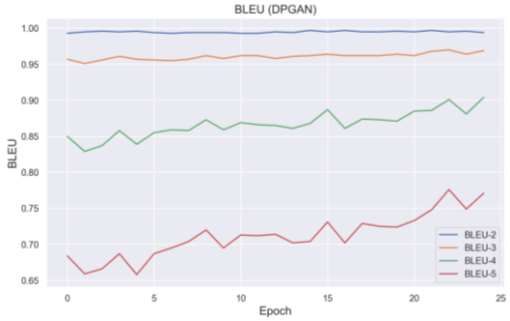

Figure 8. DPGAN BLEU Scores

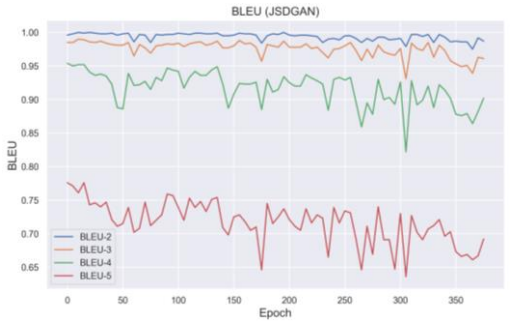

Figure 9. JSDGAN BLEU Scores

In this quantitative analysis, we also analyze the NLL-Div scores over the iterations. We found that both DPGAN and SeqGAN had a declining trend for this score as shown in Figure 10 and 11. As the iteration increases, the NLL-Div score drops while occasionally the score moves up slightly. The results indicate that the diversity of the generated code declines as the iteration goes. It is also an indicator that the generator of these two models actually assign their probability mass to a smaller region for every iteration or epoch.

NLL-Div scores from JSDGAN model are shown in Figure 12. The scores were fluctuating for every iteration, but they varied between 0.32 to 0.36. Only on one time the NLL-Div score reached higher than 0.40 and then the score fell off again to the average score. Overall, the NLL-Div scores from this model are significantly lower than the other models thus we can conclude that this model performs worst in terms of generated code diversity.

The results from MaliGAN are also interesting as the NLL-Div scores fell off at the beginning until reaching about 0.7 and then rose up to higher than 0.734 (Figure 13). The margin between the highest and the lowest was less than 0.05. Therefore, it is reasonable to conclude that MaliGAN model performs the best with regard to diversity score even though the highest NLL-Div score was obtained from DPGAN. As a note, the conclusion may change if more iterations or epochs were used.

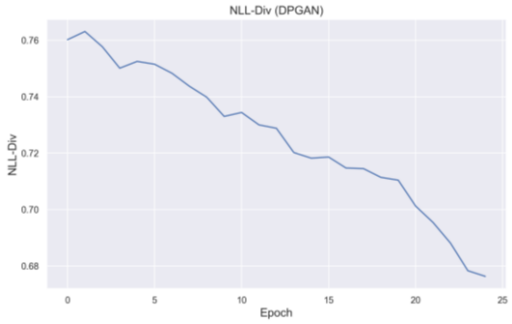

Figure 10. DPGAN NLL-Div Scores

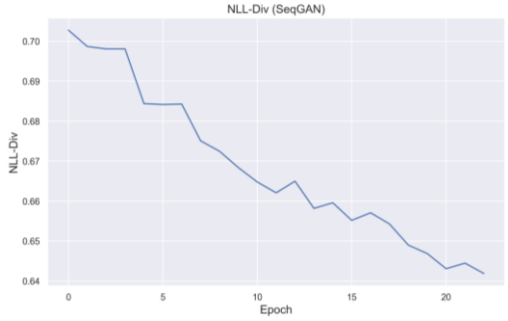

Figure 11. SeqGAN NLL-Div Scores

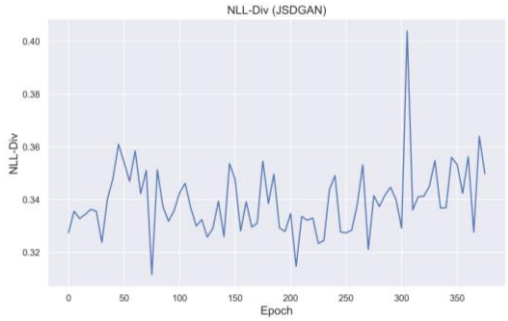

Figure 12. JSDGAN NLL-Div Scores

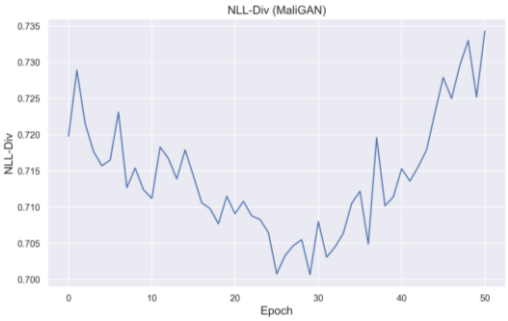

Figure 13. MaliGAN NLL-Div Scores

DOI: https://doi.org/10.29207/resti.v6i2.3940

3.2 Qualitative Results

In this part, the main priority is to observe qualitatively the generated codes so that we could find any syntax errors or interesting findings. First, we are interested to find any assertion method calls that have more or less arguments than they should have. Next, we also check whether there are any syntax error related to missing or extra special character, for example missing or double parentheses "( )", double dot "." or double comma ",". Another syntax error that we are interested to check is any misplaced special character. In this paper, we do not check whether the argument types in the method call are correct, for instance, assertIn() and assertIsInstance(). In this study, the codes in the training and testing dataset are just one line of assert method call. Thus, the variable type is unknown as there is no information about the value assigned to the variables. To simplify, we decided to omit the errors caused by incorrect type inputs.

Table 4 shows the codes generated by MaliGAN model from the iteration where the generator loss was the minimum. As shown in the table, there are 8 lines of code since the batch size is 8, and the number of samples is the same for other models in this study. By checking the errors described previously, the codes generated from this model did not contain any issues. Howver, except the code #5, the generated codes are fairly simple. The arguments are two simple values, either strings, numbers, booleans or variable names. Also, there are only four different assert methods. The code in number 5 is quite long where it calls another method with two parameters from an object or variable. Excluding assert methods, the generated codes only have two method calls (number 4 and 5).

Table 4. Codes Generated from MaliGAN

#	Generated Texts
1	assertis (status3 , true)
2	assertis (position2 . input2 , 499)
3	assertisinstance (point2 , 844)
4	asserttrue (retrieve_input ( ))
5	assertisnot (index . calculate_result (output9 , ' not ') , 398)
6	asserttrue (count . test4)
7	assertis (text2 . current , false)
8	assertisnot (pass7 , i)

The codes generated from SeqGAN are more complex as shown in Table 5. The codes have a total of seven method calls outside the assert methods. This number is relatively high compared to the previous model. There is also more complexity in the method arguments. Despite that, we found that the first argument values across generated code samples are relatively similar. For example, the argument is often in the form of variable or object that calls another method. Only #3 and #4 where the arguments are just a simple variable and a simple method call, respectively. Further, we also found that code #1 in the table has an error. The assert method receives three arguments despite only accepting two parameters. This is a problem from the model as the training dataset does not have any assertNotIn() method with more than two arguments.

Similar issue is also found as shown in the code #2 from the generated codes of DPGAN model (Table 6). The assert method asssertIs() should accept two arguments, but the generated code only provides one argument. Even though there is an issue from the model, the variation is better compared to the previous models. The number of unique assert methods is 6, while the number of unique assert method from MaliGAN and SeqGAN are 4 and 5, respectively. Further, the codes also have more argument type variations. For example, there are simple arguments, such as booleans, numbers or variables, and there are more complex arguments, such as in code #1, #2, #3 and #5.

Table 5. Codes Generated from SeqGAN

#	Generated Texts
1	assertnotin (arr4 . search_result (' no time ') , ' after have for ' , false)
2	assertin (data2 . check_output (' people ' , 143) , false)
3	assertfalse (k)
4	asserttrue (get_param ( ))
5	assertisnotnone (count . add_input (' our '))
6	assertisinstance (error8 . is_complete (' some ' , 31) , index)
7	assertisnotnone (user9 . search_result ( ))
8	assertisnotnone (username . get_output ( ))

Table 6. Codes Generated from DPGAN

#	Generated Texts
1	assertis (result9 . check_result ( ) , true)
2	assertis (result4 . position2)
3	assertnotin (x . check_input ( ) , 592)
4	assertfalse (find_output ( ))
5	assertis (calculate_result (' from ') , 112)
6	assertnotisinstance (file5 , 617)
7	assertisinstance (str1 , false)
8	assertnotin (check5 , 415)

Lastly, we also analyze the generated code from the last model, JSDGAN, shown in Table 7. There is no error in the generated code, but they are in much simpler form compared to other models' generated codes. There are arguments that have similar values, for example "false" in #2, #3, #4, #5 and #8 indicating that more than half codes use the same argument values. Moreover, the arguments are just variable names and booleans, with only one exception in #6 where it contains a method without any parameter as the argument.

Table 7. Codes Generated from JSDGAN

#	Generated Texts
1	assertisnotnone (y . state7)
2	assertnotisinstance (size , false)
3	assertnotisinstance (data4 , false)
4	assertnotin (x , false)
5	assertisnot (state0 , false)
6	asserttrue (add_input ( ))
7	asserttrue (str9)
8	assertnotin (x , false)

When we analyzed the higher iterations generated code from all models, we found that DPGAN and JSDGAN

DOI: https://doi.org/10.29207/resti.v6i2.3940

produced codes containing errors. The JSDGAN codes, specifically, have significantly many codes with errors, such as double bracket or comma that is not followed by argument(s). In the latter iterations, the DPGAN model produced more similar codes and this justifies the NLL-Div score that were shown in the previous subsection (Figure 9).

3.3 Discussion

Based on the results from quantitative and qualitative analyses, we found that the metrics results correlate positively with the results from manual observations. Future studies may consider using the same metrics to analyze the generated codes from other GAN models. Nevertheless, manual judgments from human are still important since we can find interesting patterns that cannot be found from the metrics, such as the syntax errors. Instead of checking them manually, we could consider creating a script to automatically count the number of generated codes that cannot be compiled or run. To check the syntax error, future research should include contexts of each variable used.

To understand the contexts, we need to include the line of code that assigns value to the variable. Any multiline codes can be translated with "end of line" symbol or character. This means that the multiline codes will be treated as one sample instead of different samples that the past study did. Ideally, future studies should also use a dataset that contains real unit test codes. If the dataset is still not available, source codes from Github can be considered and compiled into one big dataset. The issue of this approach is that it requires a lot of efforts and time to manually find the codes in public repositories in Github or other sites.

Even with positive results from the quantitative metrics, the quality and the diversity score might be not that crucial. As long as the generated codes can check the method or class that will be tested, the diversity of the code is not that important at the end. However, in this study, the diversity and quality metrics are important to see whether GAN models can find patterns to generate valid codes with correct syntax and many variations.

4. Conclusion

This study is our first attempt to see the feasibility of using GAN to automatically generate unit test codes. Based on the results of this study, we found that GAN models can generate codes with high quality (relatively high BLEU scores, especially BLEU-2). Even though JSDGAN was not able to generate codes with sufficient variation or diversity, the other three models selected in this study were able to produce adequate diversity score as shown from NLL-Div and NLL-Gen scores. The NLL-Div scores from these three models are ranging between 0.6 to 0.75 which are twice amount of NLL-Div score from JSDGAN model.

Automatically generate unit test code using GAN is a shooting for the moon project. Our results show shows positive signs and potentials in the use of GAN for automatic unit test code generation. Yet, there are still many experiments and studies required to finally be able to generate unit test codes given other codes that will be tested. Future studies should explore conditional GAN models so that the generated outputs are not random. The next attempt is that the models receive a simple line of code, for example, method definitions that include the method's name and their arguments. From the provided input, the models need to generate appropriate assert method call using the details.

Acknowledgment

This work was supported by Telkom University research funds. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of Telkom University.

Reference

[1] P. Runeson, "A survey of unit testing practices," in IEEE Software, vol. 23, no. 4, pp. 22-29, July-Aug. 2006. https://doi.org/10.1109/MS.2006.91

[2] M. F. Aniche, T. M. Ferreira, en M. A. Gerosa, "What concerns beginner test-driven development practitioners: a qualitative analysis of opinions in an agile conference", in 2nd Brazilian Workshop on Agile Methods, 2011, vol 19, bl 22. https://www.ime.usp.br/~aniche/files/wbma2011.pdf

[3] S. Romano, D. Fucci, M. T. Baldassarre, D. Caivano, and G. Scanniello, "An empirical assessment on affective reactions of novice developers when applying test-driven development," Product-Focused Software Process Improvement, pp. 3–19, 2019. https://doi.org/10.1007/978-3-030-35333-9_1

[4] G. Fraser and A. Arcuri, "Whole Test Suite Generation," in IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 276-291, Feb. 2013. https://doi.org/10.1109/TSE.2012.14

[5] A. Panichella, F. M. Kifetew and P. Tonella, "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets," in IEEE Transactions on Software Engineering, vol. 44, no. 2, pp. 122-158, 1 Feb. 2018. https://doi.org/10.1109/TSE.2017.2663435

[6] S. Lukasczyk, F. Kroiß, en G. Fraser, "Automated Unit Test Generation for Python", in Search-Based Software Engineering, 2020, bll 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

[7] S. Katoch, S. S. Chauhan, en V. Kumar, "A review on genetic algorithm: past, present, and future", Multimedia Tools and Applications, vol 80, no 5, bll 8091–8126, Feb 2021. https://doi.org/10.1007/s11042-020-10139-6

[8] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri and J. Benefelds, "An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application," 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017, pp. 263-272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[9] I. J. Goodfellow et al., "Generative Adversarial Nets", in Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2, Montreal, Canada, 2014, bll 2672–2680. http://dx.doi.org/10.1145/3422622

DOI: https://doi.org/10.29207/resti.v6i2.3940

[10] Y. Li et al., "StoryGAN: A Sequential Conditional GAN for Story Visualization," 2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), 2019, pp. 6322-6331. https://doi.org/10.1109/CVPR.2019.00649

[11] Y. Balaji, M. R. Min, B. Bai, R. Chellappa, en H. P. Graf, "Conditional GAN with Discriminative Filter Generation for Text-to-Video Synthesis", in Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19, 7 2019, bll 1995–2001. https://doi.org/10.24963/ijcai.2019/276

[12] Y. Li, M. Min, D. Shen, D. Carlson, en L. Carin, "Video generation from text", in Proceedings of the AAAI Conference on Artificial Intelligence, 2018, vol 32. https://ojs.aaai.org/index.php/AAAI/article/view/12233

[13] G. H. de Rosa en J. P. Papa, "A survey on text generation using generative adversarial networks", Pattern Recognition, vol 119, bl 108098, 2021. https://doi.org/10.1016/j.patcog.2021.108098

[14] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta and A. A. Bharath, "Generative Adversarial Networks: An Overview," in IEEE Signal Processing Magazine, vol. 35, no. 1, pp. 53-65, Jan. 2018. https://doi.org/10.1109/MSP.2017.2765202

[15] B. Dai, S. Fidler, R. Urtasun and D. Lin, "Towards Diverse and Natural Image Descriptions via a Conditional GAN," 2017 IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2989-2998. https://doi.org/10.1109/ICCV.2017.323

[16] J. Xu, X. Ren, J. Lin, en X. Sun, "Diversity-Promoting GAN: A Cross-Entropy Based Generative Adversarial Network for Diversified Text Generation", in Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018, bll 3940–3949. http://dx.doi.org/10.18653/v1/D18-1428

[17] W. Nie, N. Narodytska, en A. Patel, "RelGAN: Relational Generative Adversarial Networks for Text Generation", in International Conference on Learning Representations, 2019. https://openreview.net/forum?id=rJedV3R5tm

[18] X. Liu, X. Kong, L. Liu and K. Chiang, "TreeGAN: Syntax-Aware Sequence Generation with Generative Adversarial Networks," 2018 IEEE International Conference on Data Mining (ICDM), 2018, pp. 1140-1145. https://doi.org/10.1109/ICDM.2018.00149

[19] L. Yu, W. Zhang, J. Wang, en Y. Yu, "SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient", in Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 2017, bll 2852–2858. https://doi.org/10.48550/arXiv.1609.05473

[20] T. Che et al., "Maximum-likelihood augmented discrete generative adversarial networks", arXiv preprint arXiv:1702. 07983, 2017. https://doi.org/10.48550/arXiv.1702.07983

[21] R. D. Hjelm, A. P. Jacob, A. Trischler, G. Che, K. Cho, en Y. Bengio, "Boundary Seeking GANs", in International Conference on Learning Representations, 2018. https://openreview.net/forum?id=rkTS8lZAb

[22] X. Zhang, J. Zhao, en Y. LeCun, "Character-level Convolutional Networks for Text Classification", in Advances in Neural Information Processing Systems, 2015, vol 28. https://proceedings.neurips.cc/paper/2015/hash/250cf8b51c773f3f8dc8b4be867a9a02-Abstract.html

[23] Z. Li, T. Xia, X. Lou, K. Xu, S. Wang, en J. Xiao, "Adversarial Discrete Sequence Generation without Explicit NeuralNetworks as Discriminators", in Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics, 16--18 Apr 2019, vol 89, bll 3089–3098. https://proceedings.mlr.press/v89/li19g.html

[24] K. O. Burtch, "The Most Common Variable Names," The Lone Coder - Pegasoft Canada, 18-Jul-2014. [Online]. Available: https://www.pegasoft.ca/coder/coder_july_2014.html.

[25] P. Kawthekar, R. Rewari, en S. Bhooshan, "Evaluating generative models for text generation". Stanford University, 2017. https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1174/reports/2737434.pdf

[26] K. Papineni, S. Roukos, T. Ward, en W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation", in Proceedings of the 40th annual meeting of the Association for Computational Linguistics, 2002, bll 311–318. http://dx.doi.org/10.3115/1073083.1073135

[27] Z. Liu, J. Wang, en Z. Liang, "CatGAN: Category-Aware Generative Adversarial Networks with Hierarchical Evolutionary Learning for Category Text Generation", Proceedings of the AAAI Conference on Artificial Intelligence, vol 34, bll 8425–8432, 04 2020. http://dx.doi.org/10.1609/aaai.v34i05.6361

[28] H. Yao, D.-L. Zhu, B. Jiang, en P. Yu, "Negative log likelihood ratio loss for deep neural network classification", in Proceedings of the Future Technologies Conference, 2019, bll 276–282. https://doi.org/10.1007/978-3-030-32520-6_22

[29] Y. Zhu et al., "Texygen: A Benchmarking Platform for Text Generation Models", in the 41st International ACM SIGIR Conference on Research & Development in Information Retrieval, Ann Arbor, MI, USA, 2018, bll 1097–1100. https://doi.org/10.1145/3209978.3210080

DOI: https://doi.org/10.29207/resti.v6i2.3940