Using GANs to generate test cases using BDD specifications

Nugawela, N. P. S. C.
Department of Computer Science and Engineering
University of Moratuwa
Moratuwa, Sri Lanka
cnugawela@gmail.com

***Abstract*—Testing is an integral part of software development. In recent times, ideologies such as TDD (Test Driven Development) and BDD (Behaviour Driven Development) have become popular.**

***Index Terms*—TDD (Test Driven Development), BDD (Behaviour Driven Development), GAN (Generative Adversarial Network), NLP (Natural Language Processing), POC (Proof Of Concept)**

I. Introduction

The writing of tests for software projects has become very important in the modern era of software development. BDD has become a new trend wherein the test case is written in a user readable manner.
In this paper we propose a novel method for generating test cases for software projects using GANs.

II. Previous Work

As I shall elaborate in the oncoming sections, there are many studies that are based on GANs. In those studies, it is usual to see the generation of images from random noise, [2] image generation using text descriptions, [5] and much more. Also there have been studies that consider the possibilities of employing GANs to cross domain fields. [3]
There have not been any previous instances of generating BDD tests using text descriptions and GANs. So what I am suggesting is a novel venture, in terms of GANs.

III. Generative Adversarial Networks (GANs)

A. What is a GAN

A GAN comprises of two neural network. One is used for model generation and the other is used for model validation. GANs are used in a variety of applications in the data science and machine learning domain. First introduced in the paper by I. J. Goodfellow, et at. titled "Generative Adversarial Nets" published in 2014. In it, they "propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G." The adversarial modeling framework can be applied directly when the models are both multilayer perceptrons. To learn the generators distribution p_g over data x, they define a prior on input noise variables $p_z(z)$ then represent a mapping to data space as $G(z;\theta_g)$, where G is a differentiable function represented by a multilayer perceptron with parameters $_g$. We also define a second multilayer perceptron $D(x;_d)$ that outputs a single scalar. D(x) represents the probability that x came from the data rather than p_g. We train D to maximize the probability of assigning the correct label to both training examples and samples from G. We simultaneously train G to minimize log(1 D(G(z))). [1]

B. Uses of GANs

The scholarly work based on GANs include, using conditional generative adversarial nets for convolutional face generation wherein the authors, by varying the conditional information provided to an extended GAN, use the resulting generative model to generate faces with specific attributes from nothing but random noise. An ideal training process for the above model would be as follows: spacing

* The generator outputs random RGB noise by default.
* The discriminator learns basic convolutional filters in order to distinguish between face images and random noise.
* The generator learns the correct bias (skin tone) and basic filters to confuse the discriminator.
* The discriminator becomes more attuned to real facial features in order to distinguish between the simple trick images from the generator and real face images. Furthermore, the discriminator learns to use signals in the conditional data to look for particular triggers in the image.

This process continues ad infinitum, until the discriminator is maximally confused. Since the discriminator outputs the probability that an input image was sampled from the training data, we would expect a maximally confused discriminator to consistently output a probability of 0.5 for inputs both from the training data and from the generator. [2] We can see that adversarial networks can be used to create useful cases of a target population and a model that would recognize the useful cases and filter them from a sample containing both useful and noise or unfit instances. Also, from this study we can see that even random noise can be used to generate useful instances.

The authors T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim have proposed a method based on generative adversarial networks that learns to discover relations between different domains (DiscoGAN). Inter domain relationships are very obvious to people, unlike it is for machines. For example, we recognize the relationship between an English sentence and its translated sentence in French.It is fair to consider and contemplate whether machines can also discriminate and also see the inter-relationships among various domains. The authors have addressed this in the form of a conditional image generation problem. In other words, authors tried to find a mapping function from one domain to the other can be thought as generating an image in one domain given another image in the other domain. The authors also note that this problem also brings an interesting challenge from a learning point of view as explicitly supervised data is seldom available and labeling can be labor intensive. Moreover, pairing images can become tricky if corresponding images are missing in one domain or there are multiple best candidates. So, the authors have resolved to automate the discovery of relations between two visual domains without any explicitly paired data. [3] From this study we can deduce that it is possible to discover the relationships between two domains using GANs and thus generate a GAN that would use inputs from one domain to generate instances in another domain. Thus, it is justifiable to assume that it is feasible to generate test cases, which are formed conforming to the rules of a programming language such as C++, Java, Python, etc. from use cases specified in natural language.

IV. Modern Software Development

Modern software development involves much more than simple writing of code. Test driven development was the first step in creating code that was more reliable and consistent. With the recent popularity of agile methodology and push towards code quality, tests and test automation has become very important for development. Today, Test driven development has recently re-emerged as a critical enabling practice of the extreme programming software development methodology. L. Williams, E. M. Maximilien and M. Vouk ran a case study of this practice at IBM. In the process, a thorough suite of automated test cases was produced after UML design. In this case study, they found that the code developed using a test-driven development practice showed, during functional verification and regression tests, approximately 40% fewer defects than a baseline prior product developed in a more traditional fashion. The productivity of the team was not impacted by the additional focus on producing automated test cases. This test suite aids in future enhancements and maintenance of this code. The case study and the results are discussed in detail. [4] From the study we can see that TDD can improve the quality of work and also can do so without impeding on the normal software development process. The current trend in software development are highly geared towards delivering quality software to clients, as quickly as possible. New methodologies such as Agile software development are being adopted in order to replace the old development methodologies such as waterfall. To accommodate this high quality and fast delivery process, it is imperative that the code is properly tested for the intended functionalities.

Modern development definitely calls for TDD, but it is not sufficient. The test cases can be readable and understandable for the developer, but apart from the developer, people within the project such as business analysts and the client. So, the next innovation within the software development testing was to change the way that the tests are written and used.

Now the move is towards the automation of tests and writing code that is much more comprehensible. Now, manual testing is discouraged and automated test writing is encouraged. The availability of free and open source tools for test automation such as Jenkins speaks volumes about the importance of test automation. Automated test cases allows for more efficiency and can immediately notify the developer in the case that the code is broken. In the case that there are manual tests instead of automated tests, the developer has to run the tests manually and see if code has been broken. This is a problem if there are too many tests to be run manually. The move now is also towards writing tests based on the client requirements. This way the developers can make sure that the client requirements are aligned with the actual development process.

V. Behaviour Driven Development (BDD)

Behaviour Driven Development is a relatively new branch of development wherein the client requirements are used as the baseline for generating the test cases. This is a solution to the problems we have introduced during the last section. The test cases are based completely on the requirements provided by the client. So the development process will be completely aligned with the client requirements. This results in tests that test for the end-user use cases rather than much more abstract conditions.

The business analysts can note down the client requirements in the following manner, highlighting their requirements so that it would be possible to discern the conditions required for user acceptance.

- GIVEN: The client's initial state.
- WHEN: The change that takes place, it can be a change made by the client or some other event that takes place which is not controlled by the client.
- THEN: The expected behaviour when the change occurs.

We can translate such requirements directly into test cases. The methodology to be followed in such a scenario is as follows;

- GIVEN: The starting condition. Initial conditions that we take are available at the start of the test.
- WHEN: The change that happens or is expected to happen. This is a disruptive event that will cause the event that will setup the expected response.
- THEN: The test scenario. The response that we are looking for after the disruption condition.

```
Feature: annotation
#This is how background can be used to eliminate duplicate steps

Background:
   User navigates to Facebook Given
   I am on Facebook login page

#Scenario with AND
Scenario:
   When I enter username as "TOM"
   And I enter password as "JERRY"
   Then Login should fail

#Scenario with BUT
Scenario:
   When I enter username as "TOM"
   And I enter password as "JERRY"
   Then Login should fail
   But Relogin option should be available
```

Fig. 1. Example of Cucumber Test Execution

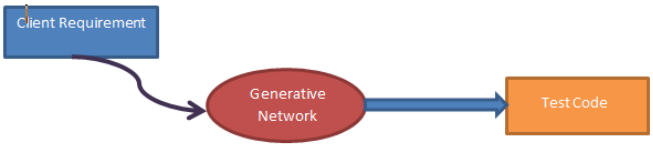

Fig. 2. The functionality of the generative network

Following Fig. 1 is an example from the cucumber BDD test case.

VI. Generation of tests using GANs

A. Justification

Generation of test cases is akin to generating images using text. Han Zhang et al. have done a study wherein they synthesize high-quality images from text descriptions as samples generated by existing text-to-image approaches can roughly reflect the meaning of the given descriptions, but they fail to contain necessary details and vivid object parts. The authors have had much success in generating images which fit the plain text descriptions of them, as specified. [5] Our case is similar in that we are using a user specified plain text to generate compilable and working test code. The success of the previous studies encourage us in that we can achieve our task of generating test cases directly from the plain text descriptions of the functionality.

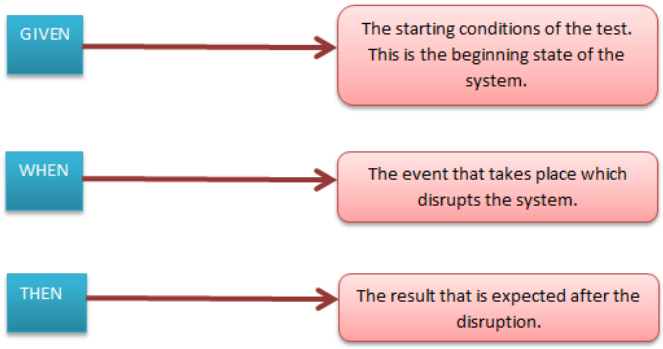

Fig. 3. BDD Test case structuring

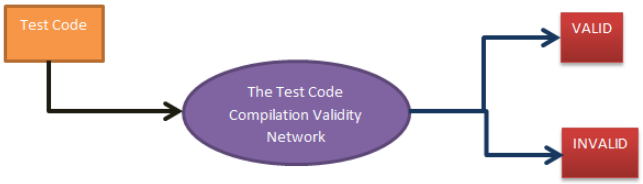

Fig. 4. Compilation validity checking network

B. Proposed Model

1) The Generative Network: As displayed by the Fig. 2, the task of the generative network is to look at the client requirement of a functionality, here we introduce the constraint that it must follow a certain format. The format being the language we use when specifying a requirement when we write down a BDD test scenario. That is, it must follow the basic structure of,

- GIVEN
- WHEN
- THEN

based on the descriptions, the generative network will be able to infer class names and their members, methods and their signatures.

2) Validation Network: The validation is done with two validation networks. Here, we have a network that checks the code compilation validity and another network that checks the validity of the generated code against the initial customer requirement.

The validation networks, as specified here, are suggestions based on the functionality and can be unified if necessary.

C. The Generative Network

The generative network is responsible for the generation of the BDD style test cases based on the client requirements given. This network should have a natural language processing capability that can reliably process the client requirement. This module must be capable of going through the requirement and deciding the necessary classes, methods and their signatures.

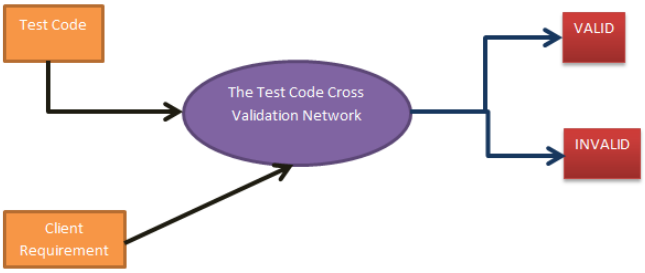

Fig. 5. Cross validation checking network

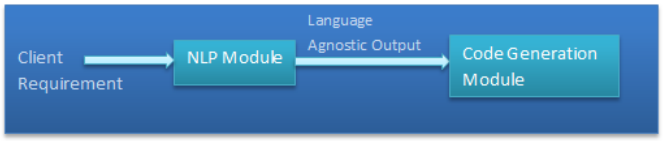

Fig. 6. Generative network modules

Also, since the test is specified in the Given-When-Then style, this module must be capable of determining the control flow of the test, quite easily as well.

The module that succeeds the Natural Language Processing (NLP) module is the module that can generate the test case based on the abstract output of the previous module. Note that we explicitly used the term *abstract* to describe the output of the NLP module. This is due to the fact that, we would like the output to be language agnostic so that we can use the output to generate code of any language. As a result, the code generation module can be programmed to generate code in any language. As a result, we can make the code generation module for generating code for Java, C++, Python, JavaScript, etc. and thus we can have multiple code generation modules rather than a single code generation module. Such a setup is useful if we have code bases in multiple programming languages, as it is sometimes practiced in organizations to have code bases in different languages but the same functionality.

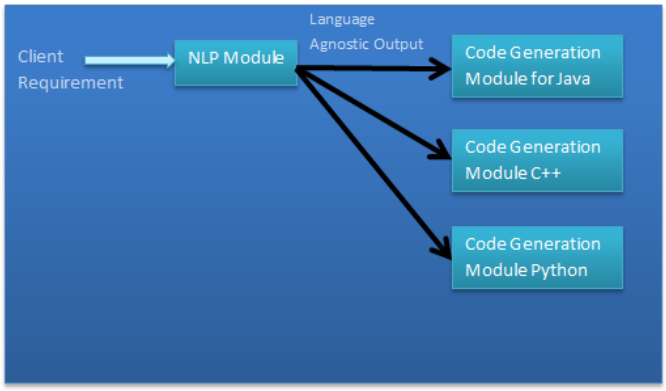

Fig. 7. Generative network modules with support for multiple languages

Fig. 8. The basic validation module

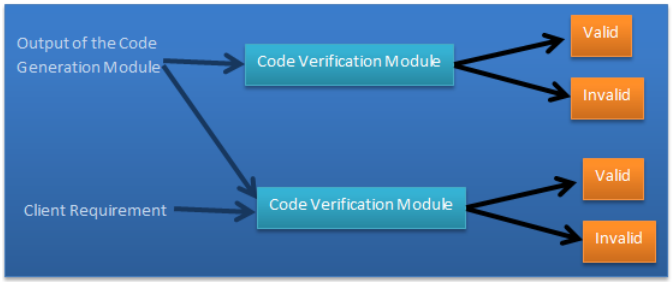

Fig. 9. The complete validation module set

D. The Discriminative Network

The discriminative network will decide whether the output of the code generation module is valid. This can be done in two ways.

1) *Code consistency checking* : Checking whether the generated code is consistent with the rules of the target programming language.
2) *Consistency with client requirement* : This is checking whether the generated code is consistent with the client requirement. It involves NLP.

Both of these are not programming language agnostic as the output of the code generation module is programming language specific. So if we need to support multiple languages, we will need multiple verification modules. Also it is advisable to have two modules to perform the aforementioned two tasks, with a module for each checking case. The complete validation module, thus achieved, is presented in the Fig. 9.

The results from the verification modules are fed back to the generative modules to improve upon their generative process, so that as time goes on, the generative procedure will improve tot the point that the verification module will not distinguish between the actual valid cases and the generated cases that are fed to it.

The code only verification module of the validation module set acts as the basic sanity test the generative network has to pass. Actually, we can first train with the code only verification module as the sole component of the discriminative network and then after it is satisfied, move onto the code and client requirement cross validation module.

E. Combined system

The combined system, comprising of the generative network and the adversarial network, will take as inputs, client requirement specifications (Written in plain text, adhering to the BDD principles.) and generate compilable/interpretable test cases, in a target programming language or languages.

The adversarial network attempts to reject these generated test cases. As the GAN evolves with time, the adversarial network becomes more and more confused between the generated test cases and true manually written test cases, ending when it cannot distinguish between manually written test cases and generated test cases.

VII. Training

For the training of the GAN, we use client requirements alongside with proper test cases written to represent them in our desired language. Here, a single language is specified because we would have to train separately to support for multiple languages. Here we would only consider the case where a single programming language is present.

Concerning the classes and methods that are available to generate the tests, we can train the neural networks on the available classes and their structure. Or we can allow the network to generate the class stubs based on the client requirements. Only the discriminative network needs to be attuned to the available code base as it can tune the generative network to use the correct classes, members and methods. But it is wise to tune the generative network in advance in order to minimize training times. Apart from the developer created classes, the GAN must also learn the basic data types of the language such as int, float, etc. and language provided data structures such as vectors, arrays, maps, etc. This is further more complicated when the developers use 3^{rd} party libraries, as we have to train for their specifics as well. So as to simplify matters, the first iteration can cover a subset of tests that are lower in complexity. The NLP module is used to extract the relevant tags that are useful to us from the plain text client requirement specification. It can be a neural network or otherwise. It is also notable that it is not imperative to have the NLP module as we can make the code generation module absorb that complexity as well. But, we can benefit hugely from having a separate NLP module for tag extraction as it is reusable, so therefore, it is worthwhile to have the NLP module.

The discriminative model testing plan is done via feeding the valid test cases as input to the verification module (That discriminates between valid and invalid test case as far as programming language consistency is concerned. i.e. the code only verification module) as valid inputs during feedback and feeding invalid test cases as invalid inputs. (during feedback) This will train the discriminative network to distinguish between the valid and invalid. We also have to train the client requirement cross validation module in a similar fashion as well. Here, we have 3 combinations.

1) *The test case is consistent with the client requirement* : Valid state is fed back to the network.
2) *The test case is not consistent with the client requirement* : Invalid state is fed back to the network.
3) *The client requirement is inconsistent/incoherent* : Invalid state is fed back to the network.

Here, we consider that if the test case itself is inconsistent with the programming language falls under the 2^{nd} scenario. The 3^{rd} case deals with the scenario that the specified client requirement is wrongly stated. This can be due to meaningless client requirements, client requirements not being specified according to the guidelines, badly specified requirements, etc.

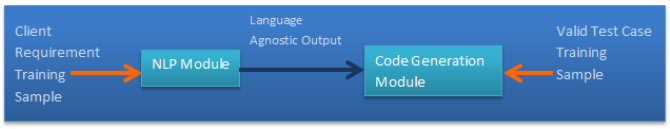

Fig. 10. Generative model training plan

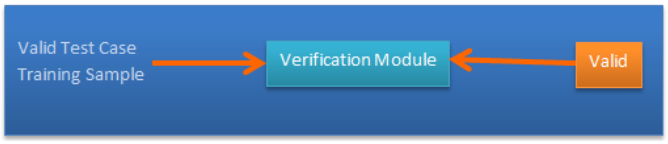

Fig. 11. Discriminative model valid instance training plan

VIII. Suggestions on implementation

The first iteration of the GAN can be formed for simple cases such as basic arithmetic operations, simple algorithmic tasks and the like to train a basic GAN as a Proof Of Concept (POC) exercise. We also suggest adhering to structure of GIVEN, WHEN and THEN structure of test scenario writing within the client requirement specifications. Also, it will be practical to have only one target programming language for this first iteration.

IX. Noted improvements

Client requirement processing can be extended to include spell and grammar correction to try to form meaningful requirement specifications. Also we can expand the code base of the GAN's neural network so it can form much more complex test cases.

X. Conclusions

In the modern software development ecosystem it is imperative to deliver high quality software products to the clients efficiently. TDD and lately, BDD have helped tremendously to achieve these goals. The problem of generating test cases for user specified scenarios using GANs is a novel and a promising one which can help organizations and individuals to develop better code. Here we have outlined a proposal for generating BDD test cases automatically using GANs. We have to specify a target language for the test cases and also provide training data which consists of labeled valid and invalid client requirement specifications and their respective BDD test cases written in the target language. After training,

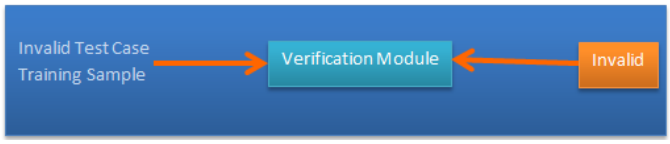

Fig. 12. Discriminative model invalid instance training plan

we would be able to generate test cases for valid client requirement specifications using the generative neural network.

REFERENCES

[1] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. C. Courville, and Y. Bengio, "Generative adversarial nets." In Proceedings of NIPS, pages 2672 2680, 2014

[2] J. Gauthier. "Conditional generative adversarial nets for convolutional face generation." *Class Project for Stanford CS231N: Convolutional Neural Networks for Visual Recognition, Winter semester,* 2014(5):2, 2014.

[3] T. Kim, M. Cha, H. Kim, J. Lee, and J. Kim. "Learning to discover cross-domain relations with generative adversarial networks." International Conference on Machine Learning, 2017.

[4] L. Williams, E. M. Maximilien and M. Vouk, "Test-driven development as a defect-reduction practice", 14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003., 2003, pp. 34-45. doi: 10.1109/ISSRE.2003.1251029

[5] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. Metaxas, "Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks," arXiv preprint arXiv:1612.03242, 2016.