크래시된 이미지와 딥 클러스터링을 통한 크래시 분류 개선

김요한*, 이상준**
*제주대학교 컴퓨터공학과
e-mail : patori@naver.com

Improving crash classification with crash image and deep clustering

Yo-Han Kim, Sang-Jun Lee
Dept. of Computer Engineering, Jeju National University

요 약

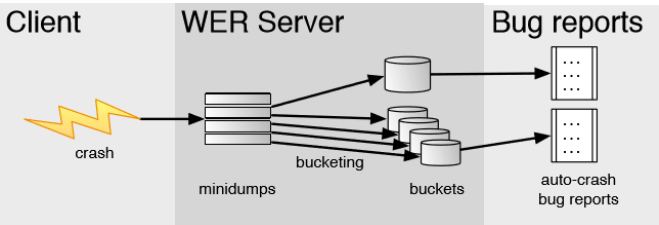
소프트웨어 크래시 분류를 개선하기 위해 호출 스택 정보를 기반한 많은 연구들이 있다. 본 연구에서는 크래시 직전 이미지를 수집하여, 기존 호출 스택 기반의 분류에서 발생하는 문제를 개선하고자 한다. 또한 이미지 자체의 직관성으로 개발자뿐만 아니라 개발 지식이 없는 실무자도 크래시 정보를 활용할 수 있고, 문제 해결을 위한 재현 루트 파악, 위변조 여부와 같은 추가 정보를 확인할 수 있을 것으로 기대한다. 비지도 학습 기반인 딥러닝 클러스터링 N2D 알고리즘을 통하여 이미지를 자동 분류하고 순위화하는 시스템을 구축하여, 특정 소프트웨어에 특화되지 않고 다양한 소프트웨어의 크래시 이미지 자동 분류에 기여할 수 있을 것으로 기대한다.

1. 서론

소프트웨어 크래시는 가장 심각한 결함 중 하나이며, 반드시 수정해야 할 정도로 해결을 위한 우선순위가 높다. 크래시 해결을 용이하게 하기 위해 크래시 발생시 사용자로부터 오류 보고서를 자동으로 수집하기 위해 Windows Error Reporting [1], Mozilla Crash Stats [2], Apple Crash Report [3]와 같은 많은 오류 보고 시스템이 배포되었다. 오류 보고서에는 크래시 지점의 모듈 이름 및 호출 스택과 같은 정보가 포함된다. 이러한 정보는 크래시 원인을 파악하려는 소프트웨어 개발자에게 매우 유용하다. 그러나 경우에 따라 많은 오류 보고서가 매일 도착할 수도 있다. 이러한 오류 보고서 중 다수는 실제로 동일한 버그로 인해 발생하는 중복된 보고서이다. 개발자의 디버깅 작업을 줄이려면 중복된 오류 보고서를 정확하게 분류하는 것이 매우 중요하다.

대표적인 오류 보고 시스템으로는 Microsoft WER (Windows Error Reporting)[1] 시스템이 있다. WER 시스템에서 오류 보고서는 (그림 1)과 같이 "버킷(bucket)"이라고 하는 분류 단위로 분류된다. 크래시 유형은 코드 오프셋, 빌드 버전, 시스템 에러 코드 등을 기준으로 패턴을 분석하여 버킷으로 분류된다.

크래시 분류에 있어서 크게 두가지 문제점에 대한 해결 방안을 찾기위한 연구가 주로 진행 되고 있다. 하나는 서로 다른 원인으로 발생한 버그를 동일 버그로 분류 하는 문제이고, 다른 하나는 이와 반대로 동일한 버그를 다르게 분류하는 (Second-bucket problem) 문제이다. 이러한 문제를 해결하기 위하여, 분류 성능을 개선하기 위한 연구가 다양하게 진행되고 있다.



(그림 1) WER 오류 보고 시스템의 개요

분류 성능 개선을 위한 기존 연구로는 "Crash Graphs 를 통한 분류 방법 개선" [4]이 있다. 이 연구에서 호출 스택을 기반으로 그래프를 만들고, Graph 간의 유사도를 평가하여 서로 다른 유형으로 분류된 크래시를 재분류한다.

또 다른 연구로는 호출 스택 최상위에서 호출 함수의 텍스트 거리를 계산하여 유사성을 비교하는 연구 [5]가 있다. 최상위 스택만으로 서로 다른 버그를 동일 버그라고 잘못 분류하는 경우에 대한 연구로서 크래시 유형별로 부분 그룹화를 수행하여 호출 스택 최상위에서 확인이 되지 않는 크래시를 예측한다.

최근에는 크래시 분류에 대한 연구는 호출 스택관련 텍스트 기반의 머신러닝 자동 분류 관련 연구가 활발하게 이루어지고 있다.[6][7] 그러나 호출 스택을 기반한 정보만으로는 정확한 오류 위치를 발견하기는 매우 어려울 수 있다.

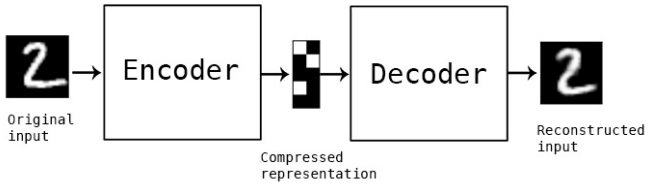
본 논문에서는 호출 스택 기반 분류의 한계점을 확

인하고, 크래시 직전 이미지를 수집하여, 이미지 기반 자동 분류를 하려고 한다. 우선 호출 스택별로 크래시 유형을 분리한다음, 오토인코더(autoencoder)기반인 N2D[8] 딥러닝 이미지 클러스터링 알고리즘을 사용하여, 이미지 유사성을 판단하여 클러스터링하고, 클러스터링 결과가 높은 순서로 순위 분류를 진행한다. 이미지 유사성을 기반한 분류로 주로 어떤 상황에서 충돌이 자주 발생하는지 확인할 수 있고, 호출 스택 텍스트 정보만으로 확인이 어려운 부분을 찾아낼 수 있는 결과를 확인한다.

2. 배경이론

1) 오토인코더 (Autoencoder)

오토인코더[9]는 두 가지 주요 구성 요소로 구성된 심층 신경망이다. 하나는 인코더(encoder)로, 입력값 x를 새로운 특징벡터(h = f(x))로 매핑하는 함수를 학습한다. 두번째 구성요소는 학습 된 특징 공간을 원래 입력공간 (r = g(h))로 다시 매핑하는 기능을 학습하는 디코더(decoder)이다. 오토인코더는 인코더 단계에서 디코더 단계를 거치면서 손실이 발생하는데, 이 손실을 최소화 하는 방향으로 학습한다.



(그림 2) 오토인코더 과정

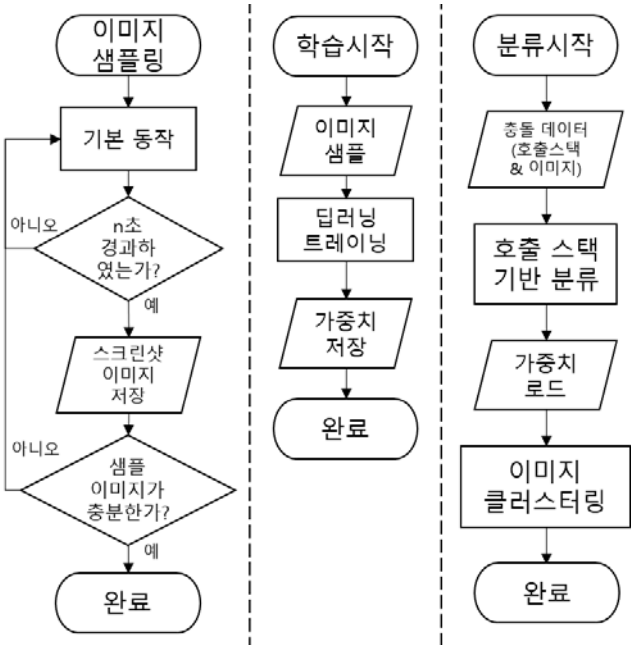
2) N2D (Not Too Deep) 알고리즘

N2D 알고리즘은 2019 년도에 제안된 딥 클러스터링 알고리즘의 하나로 MNIST 필기체 숫자 이미지 클러스터링에서 정확도 0.979 로 우수한 결과를 보였다. 오토인코더 기반으로 차원 압축을 진행하여 특징점을 추출하는데, 중간 표현 계층을 간소화하고, 그 결과를 UMAP[10]과 Gaussian Mixture Models(GMM)[11]을 이용하여 클러스터링하는 알고리즘이다. 기존 알고리즘에 비해서 학습 속도가 빠른 장점이 있다.

3. 시스템 구현

시스템 구현은 (그림 3)과 같이 크게 세가지 단계로 분류된다. 첫번째는 학습을 위한 샘플 수집 단계로 기본 이미지 특성을 추출하기 위한 이미지 정보를 수집한다. 두번째는 학습 단계로 오토인코더를 이용하여 학습하고, 학습 데이터 가중치를 저장한다. 세번째는 자동 분류 단계로 크래시 발생시 전달된 호출 스택 정보를 가지고 분류를 한 뒤 이미지를 클러스터링 한 뒤, 순위화 단계를 거쳐서 자동 분류를 진행한다. 각 단계에 대한 상세 구현 방법은 다음과 같다.

샘플 수집 단계: 본 논문에서는 이미지 분류를 보다 직관적으로 쉬운 예시로 확인하기 위해 이미지 확인이 명확한 'Super Mario Bros'[12] 오픈 게임 소스를 활용하여, 크래시 덤프 및 호출 스택 정보 및 스크린샷 이미지를 생성하였다. 딥 클러스터링 알고리즘 적용을 위해서는 많은 양의 학습 데이터가 필요한데, 게임 기본 동작을 진행하고 매 3 초마다 스크린샷 이미지를 생성하여 샘플을 추출하였다. 이와같이 기본 동작 데이터를 통하여, 초기에 수집된 크래시 정보가 없더라도 학습이 가능하여 기반 시스템 구축이 가능하다.



(그림 3) 크래시 분류 시스템 구현 흐름도

학습 단계: 딥 클러스터링 알고리즘 중 N2D[8] 알고리즘을 적용하여 구현하였다. 해당 알고리즘을 사용하여 보다 적은 샘플이미지와 시간으로 좋은 클러스터링 결과를 얻을 수 있었다. 테스트를 위한 샘플이미지의 크기는 800 x 448 였지만, 방대한 크기로 인한 파라미터 증가로, 학습시간이 기하급수적으로 늘어난다. 본 연구에서는 학습시 메모리 부족으로 400x224로 입력값을 축소하여 적용하였다.

Layer (type)	Output Shape	Param #
input (InputLayer)	(None, 224, 400, 3)	0
flattened_input (Flatten)	(None, 268800)	0
encoder_0 (Dense)	(None, 400)	107520400
encoder_1 (Dense)	(None, 10)	4010
decoder_1 (Dense)	(None, 400)	4400
decoder_0 (Dense)	(None, 268800)	107788800
reshape_2 (Reshape)	(None, 224, 400, 3)	0

Total params: 215,317,610
Trainable params: 215,317,610
Non-trainable params: 0

<표 1> Autoencoder 신경망을 적용한 결과 파라미터

Autoencoder 를 적용 내용은 <표 1>과 같다. 최적화 프로그램은 'Adam'을 적용하였으며, 모든 계층은 'ReLU' 활성화 함수를 사용하였다. 중간 결과를 활용해야 하기 때문에 encoder 마지막 단계에서 차원수를 10 개로 줄여 학습 하였다. 이미지 학습은 epochs = 100 으로 설정하고 진행하였다. Epochs 를 1000 으로 설정하였으나 오히려 손실률이 커지는 상황이 발생하여 학습 주기를 줄였고, 손실률 0.04 정도로 만족할 만한 수준을 얻을 수 있었다.

자동 분류 단계: 분류를 하기위해서는 우선 크래시를 발생시켜야 한다. 크래시를 강제로 발생시키기 위해 캐릭터와 적 오브젝트간의 접촉시 간헐적으로 크래시를 발생시키는 코드를 <표 2>와 같이 작성하였다.

```
void Map::UpdateMinionsCollisions()
{
  … // 충돌 체크 성공시 크래시 발생
    int crashPoint = rand() % 2
    if (crashPoint == 0)
    {
        Minion[i][j] = nullptr;
        Player = nullptr;
    }
}
```

<표 2> 충돌시 크래시 원인을 제공하는 코드

<표 2>와 같이 작성한 코드는 문제되는 객체 포인터를 참조하는 시점에서 널 포인터 체크를 하지 않으면, 크래시가 발생한다. 이와 같은 크래시 유형은 객체를 참조하는 시점과 잘못된 소스코드가 있는 위치가 일치 하지 않기 때문에 호출 스택 기반 분류로는 문제 발생 코드 위치 확인이 어려워 오브젝트가 널 포인트가 된 지점을 직접 디버깅해보기 위해 재현을 해야 하는 상황이 발생 할 수 있다.

오류 관련 정보를 수집하기 위해서는 소스 코드에서 예외 핸들러를 등록하고, 예외 핸들러에서 덤프 및 호출스택 정보, 스크린샷 이미지 정보를 저장하도록 하고, 크래시를 발생시킨다. 일정량의 크래시를 확보하고, 발생한 호출 스택 정보 기반으로 기본 분류를 진행하고, 딥 클러스터링을 이용한 스크린샷 이미지 자동 분류를 통하여 그 결과를 확인한다.

이미지 평가를 위해서는 N2D 알고리즘에 따라서 학습 단계에서 저장한 가중치 정보를 이용하여, 은닉 계층 적용 결과를 추출한다. 추출된 정보는 다시 UMAP, Gaussian Mixture Models(GMM) 알고리즘을 거쳐서 보다 더 명확하게 구분되도록 클러스터링을 진행한다. 마지막으로 클러스터링된 개체수가 가장 많은 순서로 순위화를 진행한다.

4. 테스트 방법 및 평가

샘플 이미지 수집 및 학습: 총 4 스테이지를 1 회 플레이 하여 3 초마다 스크린샷을 저장하여 총 388 장의 이미지를 수집하였다. 이 이미지를 가지고 학습을 수행하였고, 클러스터수를 'c' 라고 할때, c = 10 으로 설정하고, 배치사이즈를 388, 주기를 100 으로 설정하여 학습한 결과, 손실률 0.04 정도의 수치를 보였다. 학습된 가중치는 hdf5 포맷으로 저장하였다. 학습된 가중치로 클러스터링이 잘 되는지 테스트를 진행하였고, (그림 4)과 같이 분류 되었다

(그림 4) 플레이 데이터 기반 클러스터링 결과중 일부 (c = 10 중 6 개 표시)

결과 C1 ~ C6 을 확인하였을 때, 스테이지 구분이 비교적 명확한 {C1, C3, C5, C6}은 잘 구분을 하였지만 {C2, C4}의 경우는 두개의 스테이지가 혼합된 결과를 보였다. {C1, C6}, {C2, C4}의 경우에도 서로 비슷하나, C2 가 지형 정보가 더 많아서 따로 분류되었음을 추측할 수 있었다.

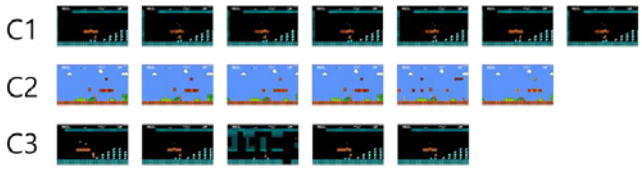
A 유형: 47 건	B 유형: 17 건
Minion::getBloockID()	Player::getXPos()
Map::DrawMinions()	Map::UpdateMinions Collisions()
Map::Draw()	MenuManager::Update()
MenuManager::Draw()	CCore::Update()
CCore::Draw()	CCore::mainLoop()
CCore::mainLoop()	SDL_main()
SDL_main()	main_getcmdline()
main_getcmdline()	WinMain()
WinMain()	

<표 3> <표 2>에서 파생된 크래시 호출 스택 유형.

호출 스택 기반 크래시 분류: <표 2>를 기반으로 총 58 건의 크래시를 발생시켰고, 그 결과 <표 3>과 같이 두개의 크래시 유형 분류가 발생하였다. 개발자가 <표 2>의 내용을 미리 알고 있다면, A 유형은 'Minion[i][j] = nullptr;' 이 원인이고, B 유형은 'Player = nullptr;' 이 원인임을 쉽게 유추할 수 있지만, A 유형의 경우만 발생 할 경우, 호출 스택만으로는 어느 과정에서 원인이 발생하였는지 유추하기 어렵다. 따라서 호출 스택 만으로 A 와 B 유형의 공통점을 찾을 수 없기 때문에 같은 위치에서 파생했을 것이라는 유추는 기존 호출 스택 기반연구에서는 확인하기 어렵다.

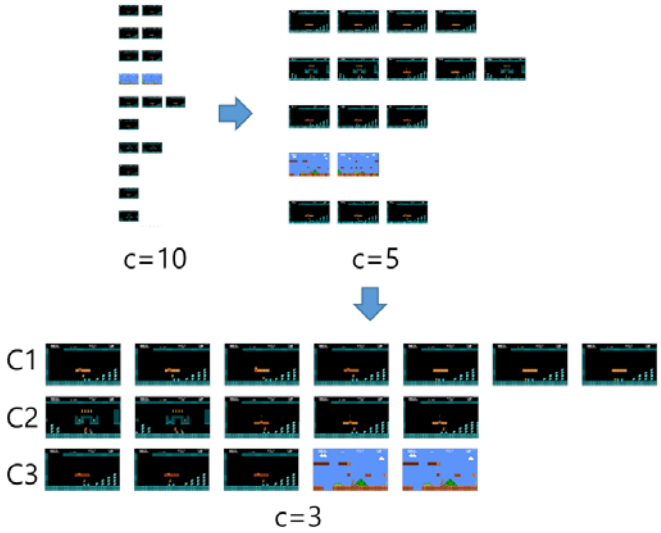
이미지 기반 분류: 호출 스택 기반으로 분류된 이미지를 이미지 클러스터링을 통하여 추가 분류한 결과는 (그림 5)와 (그림 7)과 같다. A 유형의 주요 순위 3 개만 확인하였을 때, C1 이 7 건, C2 가 6 건, C3 이 5 건으로 전체 건수 중의 38%가 분류 되었다. (그림 6)와 같이 높은 순위의 이미지들의 공통점을 분석해보

면, 캐릭터와 적과의 접촉이 있었고, 적이 둘 이상인 곳에서 크래시가 발생한 것을 짐작할 수 있다. 또한 {C1, C2}의 경우, 공통된 이미지가 많은 것으로 보아서 해당 위치에서 크래시 재현이 잘 되는 것을 유추할 수 있다. 이와 같이 이미지 분류만으로도 호출 스택에서는 부족한 상당히 많은 정보를 얻을 수 있음을 알 수 있다.



(그림 5) A 유형 2 차 이미지 분류 화면
(c = 10 중 상위 3 개 표시)

(그림 6) 대표 이미지 공통점 분석



(그림 7) B 유형의 클러스터수를 변경하여 적용한 결과 화면

B 유형의 경우에는 추출된 결과물이 총 17 건으로 c=10 으로 분류하였을 때, 클러스터당 1~3 건의 이미지가 고루 분포되어 있어서 순위 분류에 적절하지 않다. 따라서 크래시 수량에 따라 적절한 클러스터 수를 자동으로 추출하기 위해 편차를 구하여 c 값을 변경하여 다시 분류를 진행하면 결과는 (그림 7)와 같이 분류된다. 클러스터수를 3 개로 줄였을때, {C1, C2}의 내용이 전부 비슷한 내용을 표시하고 있음을 확인할 수 있다. 이것으로 A 유형과 B 유형이 모두 비슷한 지점에서 크래시가 발생하였음을 확인할 수 있고, 적과의 접촉 시점에 모두 크래시가 발생한 것을 시각적으로 추측할 수 있다.

위와 같은 분석으로 볼 때, 호출 스택 분류에서 B 유형의 'Map::UpdateMinionsCollisions()' 지점에 주 원인이 있음을 가정하고 버그를 추적 해보는 것이 보다 효율적일 것이라는 결론을 이미지 분류를 통해 유추할 수 있다. 또한 재현 및 해결을 확인하기위한 테스트를 진행 하려면 A, B 유형 공통 1 순위인 두번째 스테이지에서 진행하는게 유리하다는 것을 알 수 있다.

5. 결론

이와 같은 결과로 이미지를 활용한 자동 분류 방법이 호출 스택 분류를 보완하여 사용될 수 있음을 알 수 있다. 향후에는 연속적인 이미지 / 사운드 등의 특정 정보도 추가로 활용하는 방법도 연구가 가능하며, 클러스터링된 정보를 좀 더 의미 있는 특징이 나타나도록 순위화를 자동으로 분류하는 방법에 대한 연구가 더 필요하다.

참고문헌

[1] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: ten years of implementation and experience," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. Big Sky, Montana, USA: ACM, 2009, pp. 103-116.

[2] Mozilla, "Crash Stats", 2010, http://crash-stats.mozilla.com.

[3] Apple, "Technical Note TN2123:CrashReporter", 2010, available at : http://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html.

[4] S. Kim, T. Zimmermann, N. Nagappan, Crash graphs: An aggregated view of multiple crashes to improve crash triage, Proc. 41st International Conference on Dependable Systems & Networks (DSN), Hong Kong, June 2011, 486 – 493.

[5] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In Proceedings of the 34th International Conference on Software Engineering, pages 1084–1093, 2012.

[6] W Le, D Krutz, How to Group Crashes Effectively: Comparing Manually and Automatically Grouped Crash Dumps, 2012

[7] M Cupurdija, Evaluating methods for grouping and comparing crash dumps, 2019

[8] Ryan McConville, Raul Santos-Rodriguez, Robert J Piechocki, Ian Craddock, N2D:(Not Too) Deep Clustering via Clustering the Local Manifold of an Autoencoded Embedding, 16 Aug 2019,

[9] M. Tschannen, O. Bachem, and M. Lucic, "Recent advances in autoencoder-based representation learning," 3rd workshop on Bayesian Deep Learning (NeurIPS 2018), 2018.

[10] L. McInnes, J. Healy, and J. Melville, "Umap: Uniform manifold approximation and projection for dimension reduction," arXiv preprint arXiv:1802.03426, 2018

[11] D. A. Reynolds, "Gaussian mixture models," in Encyclopedia of Biometrics, 2009.

[12] https://github.com/jakowskidev/uMario_Jakowski