석사학위논문

이미지 딥 클러스터링을 통한 크래시 이미지 분류 기법

제주대학교대학원

컴퓨터공학과

김요한

2020년 2월

이미지 딥 클러스터링을 이용한 크래시 분류 기법 (Crash classification using image deep clustering)

지도교수 이 상 준

김 요 한

이 논문을 공학 석사학위 논문으로 제출함

2019 년 12 월

김요한의 공학 석사학위 논문을 인준함

심사위원장 _____㊞

위 원 _____㊞

위 원 _____㊞

제주대학교 대학원

2019 년 12 월

요약

소프트웨어 크래시는 가장 심각한 소프트웨어 시스템 결함 중 하나이며, 반드시 수정해야 하는 우선순위가 높은 문제이다. 따라서 빠른 대응과 디버깅을 하기 위해서는 크래시 발생 시 자동으로 수집하기 위한 크래시 리포트 시스템을 구축하여 대응해야 한다. 수집된 크래시 리포트 중 다수는 동일 버그로 인하여 발생하는 중복된 리포트이다. 따라서 개발자가 디버깅 작업을 줄이고 문제 파악을 빠르게 하기 위해서는 크래시 리포트를 정확하게 분류하는 것이 중요하다.

본 연구에서는 소프트웨어 크래시 직전 이미지를 활용하여 유사한 이미지끼리 자동 분류하는 기법을 제안하였다. 같은 호출 스택이어도 다른 원인을 제공하는 경우 및 원인은 같지만 다른 호출 스택으로 분류되는 문제에 대하여 직관적인 판단을 할 수 있도록 한다. 이미지 기반 분류는 이미지만으로도 크래시 당시 상황을 유추할 수 있는 많은 정보를 제공하기 때문에 개발자뿐만 아니라 개발 지식이 없는 다른 실무자들도 크래시 정보를 활용할 수 있고, 문제 해결을 위한 재현 루트 파악, 위변조 여부와 같은 추가 정보를 크래시 직전 이미지를 통하여 확인할 수 있다. 이미지를 분류하는 방법은 비지도 학습 기반인 딥러닝 클러스터링 알고리즘을 통하여 크래시 된 이미지에 대하여 자동 분류를 수행하고, 클러스터링 된 결과를 순위화하여 빠르게 판단할 수 있도록 정보를 제공한다. 제안한 기법은 크래시 이미지 분류를 위하여 초기 딥러닝 학습이 필요한데, 이를 해결하기 위해 간단하게 표본 이미지를 수집하는 방법에 대하여 제시하였고, 이를 활용하여 특정 소프트웨어에 특화되지 않고 다양한 소프트웨어의 크래시 이미지를 분류할 수 있다.

Abstract

The software crash is one of the most serious software system faults and is a high priority issue that must be corrected. Therefore, for fast response and debugging, a crash report system must be established to automatically collect when a crash occurs. Many of the collected crash reports are redundant reports that result from the same bug. Therefore, it is important for developers to correctly classify crash reports in order to reduce debugging and speed problem identification.

In this study, we proposed a technique to automatically categorize similar images among others using images just before a software crash. If the same call stack also provides different causes, and for problems that have the same cause but are classified as different call stack, an intuitive judgment can be made. Image-based classification provides a lot of information to infer the situation at the time of the crash, so not only developers but also other practitioners without development knowledge can utilize the crash information, and further information such as identifying a reproducible route for troubleshooting and whether to falsify can be viewed through the image immediately before the crash. The method for sorting images provides information so that automatic classification of cracked images can be performed through deep learning clustering algorithm, which is based on non- map learning, and that clustered results can be ranked and judged quickly. The proposed technique required initial deep learning for the classification of crash images, which is presented simply as to how to collect sample images, which can be used to classify the crash images of various software without being specific to specific software.

목 차

그 림 목 차

표 목 차

I. 서론

1. 연구의 배경 및 목적

소프트웨어 크래시는 소프트웨어 시스템에 있어서 가장 심각한 결함 중 하나이며, 반드시 수정해야 할 정도로 해결을 위한 우선순위가 높다. 크래시를 쉽게 해결하기 위해서는 크래시 발생 시 사용자로부터 크래시 리포트를 자동으로 수집하는 시스템이 필요하고, Windows Error Reporting[1], Mozilla Crash Stats[2], Apple Crash Report[3]와 같은 많은 크래시 리포트 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.

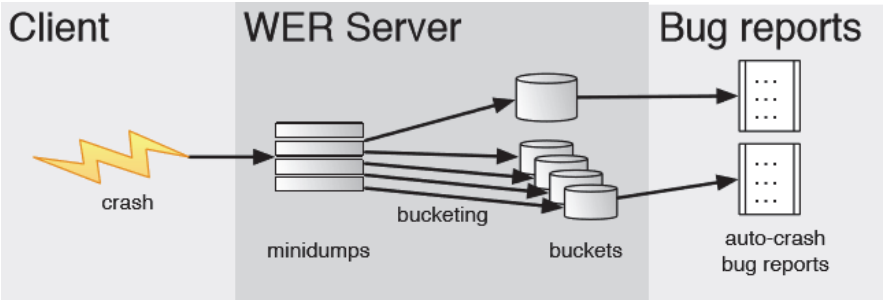

Figure 1. Overview of the WER Error Reporting System

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

크래시 분류에 있어서 크게 두 가지 문제점이 있다. 하나는 서로 다른 원인으로 발생한 버그를 동일 버그로 분류하는 문제이고, 다른 하나는 이와 반대로 같은

버그를 다르게 분류하는 문제(Second-bucket problem)이다. 이러한 문제를 해결하기 위하여, 분류 성능을 개선하기 위한 연구가 다양하게 진행되고 있다.

기존 연구에서는 주로 호출 스택과 버전, 빌드 시간, OS 예외코드, 시스템 에러코드, 코드 오프셋 등 텍스트 정보를 이용하여 분류하였다.

본 논문에서는 분류 문제를 해결하기 위한 기존 연구에서의 호출 스택 기반 분류의 한계점을 확인하고, 크래시 직전 이미지를 수집하여, 이미지 기반 자동 분류를 적용하여 그 효과를 확인하려고 한다. 우선 호출 스택별로 크래시 유형을 분리한 다음, 오토인코더(autoencoder)[4]기반인 N2D[5] 딥러닝 이미지 클러스터링 알고리즘을 사용하여, 이미지 유사성을 판단하여 클러스터링하고, 클러스터링 결과가 높은 순서로 순위 분류를 진행한다. 이미지 유사성을 기반으로 하기 때문에 주로 어떤 상황에서 크래시가 자주 발생하는지 확인할 수 있고, 호출 스택 텍스트 정보만으로 확인이 어려운 부분을 찾아낼 수 있는 결과를 확인할 수 있음을 확인하여 크래시 이미지 자동 분류 기법을 제안하고자 한다.

Ⅱ. 관련 연구

본 장에서는 크래시 덤프 관련 기존 연구의 해결 방법과 딥 클러스터링 관련 이미지 연구에 대하여 고찰한다.

2. 관련 연구

크래시 덤프 분류 성능 개선을 위하여 다음과 같은 문제에 관한 관련 연구들이 있다.

2.1. 같은 유형을 서로 다른 유형으로 분류하는 문제

같은 유형을 서로 다른 유형으로 분류하는 문제는 과도한 분류를 발생시키며, 크래시 덤프 분석을 위한 버그 우선순위를 선정하기 어려운 문제가 발생할 수 있다. 이와 같은 문제에 사례로 한가지 예를 들자면, ‘writeBuffer()’라는 함수에서 특정 배열 변수에 내용을 입력하는데, 배열 범위를 초과한 메모리 영역에 값을 입력했다고 했을 때, 크래시가 시점은 실제 배열을 참조하는 함수이다. 그러므로 이 배열을 참조하는 함수가 많다고 했을 때, 크래시 발생 시점의 최상위 호출 스택은 서로 다른 함수를 가리킬 수 있다. 이럴 때 같은 유형을 다른 유형으로 분류하는 문제가 발생할 수 있다.

이 문제를 해결하기 위한 대표적인 연구로는 “Crash Graphs를 통한 분류 방법 개선”[6]이 있다. 이 연구에서는 모든 크래시 유형에 대하여 Crash Graph를 만들고 유사도를 비교하는 방식으로 재분류를 하는 방법을 제안하였다. 그래프를 생성하기 위해 우선 호출 스택을 프레임(frame) 단위를 노드 단위로 정하고, 프레

임 간 호출 관계를 간선으로 표현한다. 크래시 호출 스택 순서가 A→B→C→D라고 한다면, 각각의 호출 스택 유형은 단방향 그래프로 표현된다. 그 후 각 노드 간의 관계를 두 개의 노드와 하나의 간선으로 연결된 최소 그래프 단위로 분해(Decomposing)하면 A→B, B→C, C→D 3개의 그래프를 구할 수 있다. 각각의 크래시 그래프에서 최소단위로 분해된 그래프 정보를 종합하여, 크래시 그래프를 생성한다. 크래시 그래프를 생성하는 과정은 Figure 2와 같다.

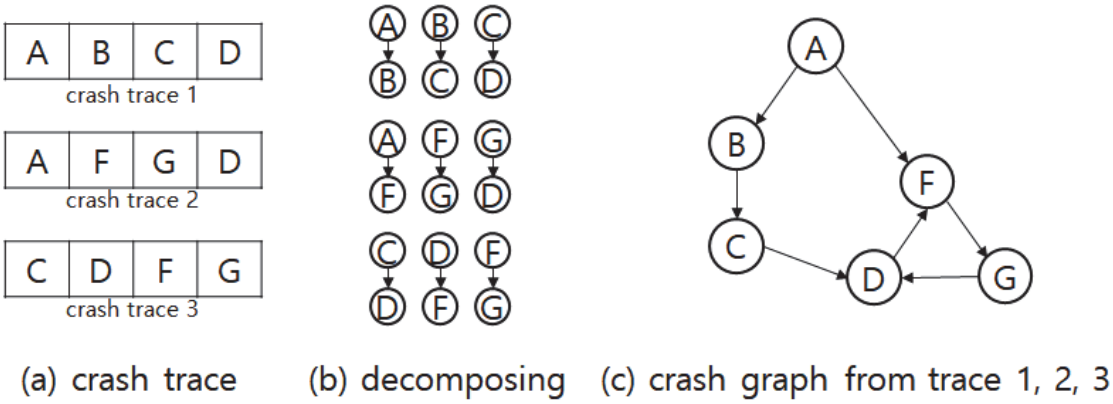

Figure 2. Example of crash graph with multiple call stacks

각 크래시 유형별로 크래시 그래프를 생성한 뒤, 그래프 간 유사성을 평가하여 서로 다른 유형으로 분류된 크래시를 재분류한다. 유사성 평가는 다음과 같은 방식으로 진행한다. 비교를 위한 크래시 유형에 대하여 각각 G_1, G_2의 크래시 그래프를 구하였을 때, 유사성 평가 방법은 식(1)과 같다.

$$Sim(G_1, G_2) = \frac{|E_1 \cap E_2|}{min(|E_1|, |E_2|)} \quad (1)$$

여기서, E는 그래프 G 간선의 집합이다. 위 수식으로 유사성을 판단하여, 다른 유형으로 분류된 같은 유형의 크래시를 재평가할 수 있다.

2.2. 서로 다른 유형을 같은 유형으로 배치하는 문제

서로 다른 유형을 같은 유형으로 배치하는 문제는 같은 유형으로 판단하여, 개

발자가 하나의 유형으로 잘못 인식하여 모든 문제를 수정하지 못하는 문제가 발생할 수 있다. 이 문제 유형의 예를 들면, 문자의 개수를 검사하는 'strlen()' 함수를 호출하여 크래시가 발생하였다고 가정하였을 때, 문자열 객체가 주원인이 될 수 있다. 그러나 문자열 객체를 잘못 조작한 함수가 여러 개라고 가정한다면, 문제 원인은 여러 개가 될 수 있으나, 호출 스택의 최상위 내용이 'strlen()' 함수로 같으므로 동일 유형으로 분류될 수 있다.

이러한 분류 문제를 해결하기 위한 연구로는 호출 스택 최상위로부터 호출 함수의 거리를 계산하여 유사성을 비교하는 연구[7]가 있다.

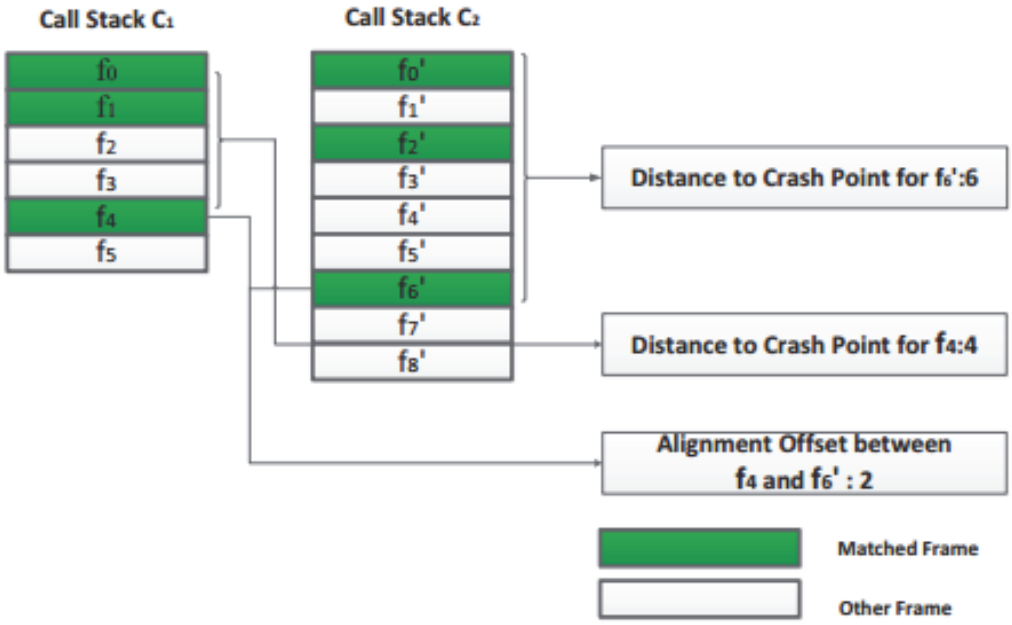

Figure 3. Call stack measurement example

최상위 스택만으로 서로 다른 버그를 동일 버그라고 잘못 분류하는 경우를 해결하는 방법에 관한 연구로서, Figure 3과 같이 호출 스택 프레임이 유사한 부분을 찾은 뒤, 각 최상위 호출 스택으로부터의 거리를 측정하여, 이 거리의 차를 이용하여 유사도를 검사한다. 각각의 크래시에 대하여 유사도를 측정하여 크래시 유형별로 부분 그룹화를 수행하여 호출 스택 최상위에서 확인이 되지 않는 크래시를 예측한다.

2.3. 그 외 연구

최근에는 크래시 분류에 관한 연구는 호출 스택 정보를 활용한 기계학습 자동 분류 연구가 활발하게 이루어지고 있다. [8][9] 그러나 호출 스택을 기반으로 한 한정된 정보만으로는 정확한 오류 위치를 발견하기는 매우 어려우므로, 크래시 리포트에 기록된 개발자 주석이나 코드 수정 이력 등의 정보를 확인하여 추가 분류를 하는 방식으로 다양한 연구가 계속 진행되고 있다. 그러나 이미지와 관련된 크래시 분류에 관한 연구는 아직 진행된 사례가 없다.

3. 배경 지식

본 연구에서 딥 클러스터링 알고리즘을 적용하기 위한 배경 설명을 설명한다.

3.1. 오토인코더(Autoencoder)

주어진 입력값에 대해 종속 변수가 없는 데이터를 학습하는 것을 비지도 학습이라고 부른다. 오토인코더는 비지도 학습 알고리즘 중 하나이다. 오토인코더는 두 가지 주요 구성요소로 구성된 심층 신경망이다. 하나는 인코더(encoder)로, 입력값 x를 새로운 특징 벡터($h = f(x)$)로 매핑하는 함수를 학습한다. 두 번째 구성요소는 학습된 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 최소화하는 방향으로 학습한다.

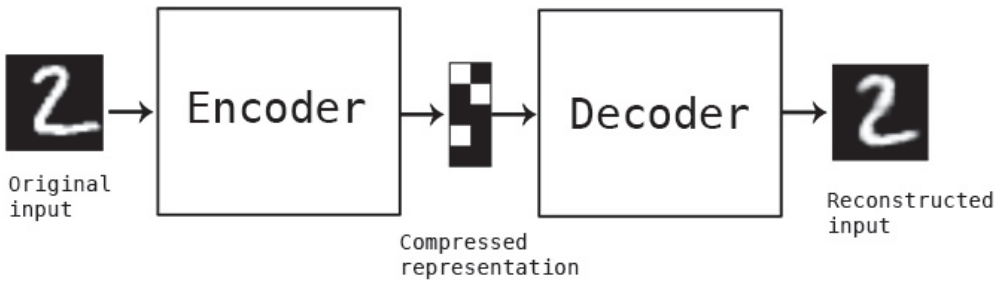

Figure 4. Auto Encoder Process

3.2. 매니폴드 학습

고차원의 데이터를 그보다 작은 차원으로 축소할 때, 그 데이터를 낮은 오류로 잘 표현할 수 있는 데이터 공간이 존재하며 이를 매니폴드(manifold)라고 한다. 매니폴드를 학습하면 원래 데이터의 정보를 잘 유지하는 동시에 데이터의 차원을 축소하여 효과적으로 데이터를 압축할 수 있고 3차원 이하로 압축하는 경우 데이터의 시각화도 가능하다. 또한, 차원이 높을수록 분포를 분석하거나 모델을 추정하는데 필요한 표본의 개수가 기하급수적으로 증가하는 문제가 발생하는데, 매니폴드 학습을 통해 차원을 줄이며 이러한 문제를 해결할 수 있다. 매니폴드 학습 방법에는 IsoMAP, t-SNE 등이 활용된다.

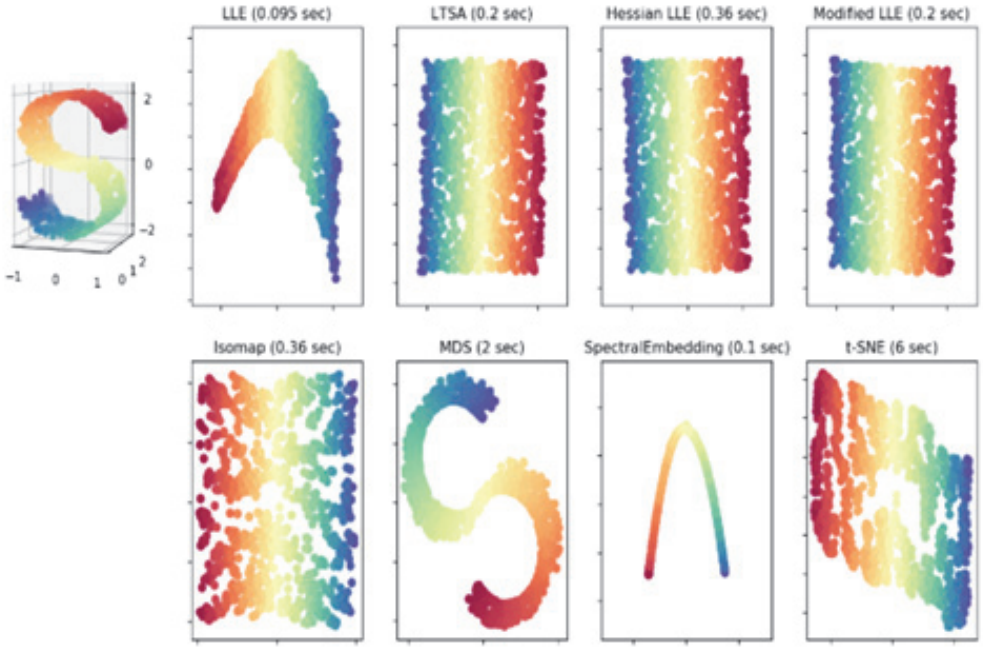

Figure 5. Manifold Learning Example

3.3. UMAP (Uniform Manifold Approximation and Projection)

UMAP[10]은 최근 제안된 매니폴드 학습법으로, 지역 구조를 정확하게 표현하려

고 하지만 글로벌 구조로도 잘 통합되는 것으로 알려져 있다. UMAP은 t-SNE와 자주 비교 된다. t-SNE는 일반적으로 큰 데이트 세트에서 어려움을 겪지만 UMAP은 확장성이 뛰어나다. 또한, UMAP은 전체 구조를 더 잘 보전하면서도, 근접 이웃과의 거리를 유지하는 데 특성화되어 있으므로, 전체와 지역적인 이점을 둘 다 가진 상태로 축소할 수 있다.

UMAP은 k-이웃 기반 그래프 알고리즘을 사용한다는 점에서 IsoMAP과 유사하다. 고차원에서 UMAP은 먼저 가중 k-이웃 그래프를 구성하고 이 그래프에서 저차원 레이아웃을 계산하게 된다. 저차원 레이아웃은 교차 엔트로피를 기반으로 가능한 원본에 대한 퍼지 토폴러지 표현과 가깝게 되도록 최적화한다.

UMAP에는 성능에 영향을 주는 중요한 하이퍼 파라미터가 있다. 첫 번째는 지역적으로 고려할 이웃의 수로 얼마나 많은 지역적인 구조가 보존되는지와 전체 구조가 얼마나 많이 수집되는지의 세분화 방법 사이의 균형을 표현한다. 로컬 구조를 통합하는 데 주로 관심이 있으므로 이웃 수에 따라 낮은 값을 선택한다. 두 번째는 대상 임베딩의 차원이다. 차원을 찾고자 하는 클러스터 수로 설정한다. 또한, 임베딩 공간의 포인트 간의 분리 허용 최소 거릿값을 설정해야 한다. 최소 거릿값이 작을수록 실제 매니폴드 구조를 정확하게 포착할 수 있으나 시각화를 어렵게 만들 수 있다.

3.4. 가우시안 혼합 모델 (Gausian Mixture Model)

가우시안 혼합 모델[11]은 Gaussian 분포가 여러 개 혼합된 클러스터링 알고리즘이다. 현실에 존재하는 복잡한 형태의 확률 분포를 Figure 6과 같이 K개의 가우시안 분포를 혼합하여 표현하는 아이디어이다.

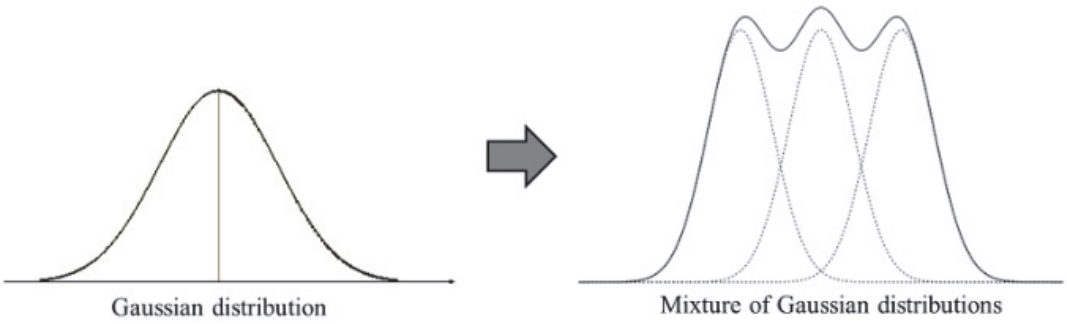

Figure 6. Mixed distribution of several Gaussian distributions

주어진 데이터 x에 대해 GMM은 x가 발생할 확률을 다음 식(2)와 같이 가우시안 확률 밀도의 합으로 표현된다.

$$p(x) = \sum_{k=1}^{K} \pi_k N(x|\mu_k, \Sigma_k) \qquad (2)$$

식(2)에서 mixing coefficient라고 하는 π_k는 k번째 가우시안 분포가 선택될 확률을 나타낸다. 따라서 π_k는 아래의 두 조건을 만족해야 한다.

$$0 \leq \pi_k \leq 1$$

$$\sum_{k=1}^{K} \pi_k = 1$$

GMM을 이용한 클러스터링은 주어진 데이터 x_n에 대하여 이 데이터가 어떠한 가우시안 분포에서 생성되었는지를 찾는 것이다.

3.5. N2D (Not Too Deep) 알고리즘

N2D 알고리즘은 2019년도에 제안된 딥 클러스터링 알고리즘의 하나로 MNIST 필기체 숫자 이미지 클러스터링에서 정확도 97.9%로 우수한 결과를 보였다. 오토인코더 기반으로 차원 압축을 진행하여 특징점을 추출하는데, 기존의 딥 러닝 알고리즘은 히든 계층을 깊게 쌓아서 복잡한 문제를 해결하려고 하는 데 비해, N2D 알고리즘은 이와 반대로 중간 표현 계층을 간소화하고, 그 결과를 입력 데이터로 사용하여 UMAP과 가우시안 혼합 모델(GMM)을 이용하여 클러스터링하는

알고리즘이다. Figure 7과 같이 딥 러닝 알고리즘과 딥 러닝이 아닌 알고리즘을 혼합함으로써 기존 딥 클러스터링 알고리즘보다 학습 속도가 빠르고 효과가 빠른 장점이 있다.

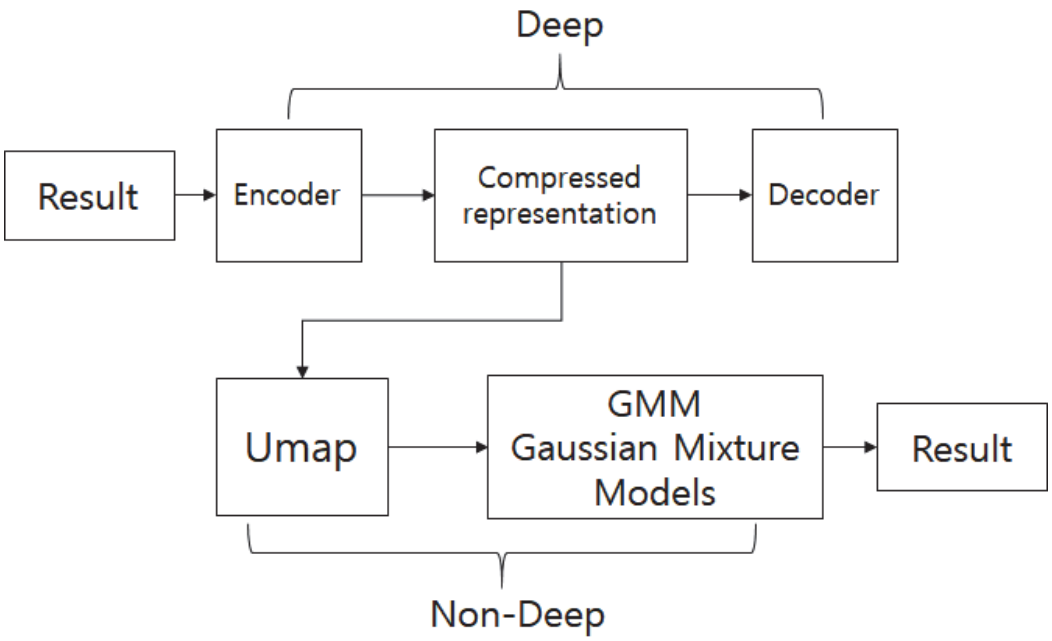

Figure 7. N2D algorithm

Ⅲ. 이미지 분류 시스템 설계

3장에서는 2장의 이론적 배경과 딥러닝 기술을 통해 진행할 구체적인 실험 방법에 대해 기술한다.

크래시 이미지 분류 시스템 프로세스는 Figure 8과 같이 크게 세 가지 단계로 분류된다.

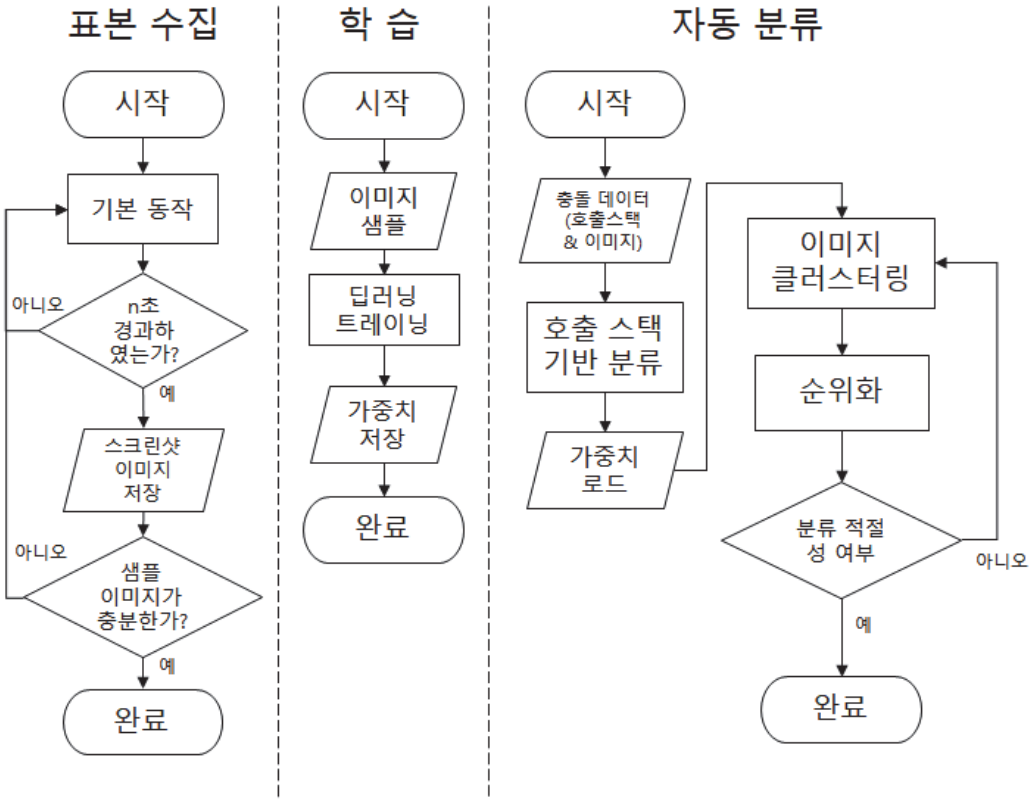

Figure 8. Crash Image Classification System Process

첫 번째 단계는 학습을 위한 자료수집 단계로 대상이 되는 소프트웨어의 기본적인 이미지 특성을 추출하기 위하여 이미지 정보를 수집한다. 두 번째는 학습 단계로 오토인코더 비지도 학습 모델을 이용하여 표본 이미지를 학습하고, 손실률을 최적화하고, 압축된 표현(Compressed representation) 층의 정보를 저장한다. 세 번째는 자동 분류 단계로 크래시 덤프 발생 시 호출 스택과 크래시 발생 당

시 스크린 캡처 이미지를 별도로 저장한다. 저장된 호출 스택 정보를 가지고 분류를 진행한 뒤, 호출 스택 분류 별로 이미지 클러스터링을 진행한다. 클러스터링이 완료된 후, 클러스터링 결과를 분석하여 각 이미지 클러스터 그룹에 대한 순위화 단계를 거치면 자동 분류가 완료된다. 각 단계에 대한 상세 구현 방법은 다음 절에서 설명한다.

1. 학습 설계

학습 설계에서는 표본 수집 방법 및 학습 단계에 대하여 설계한다.

1.1. 표본 수집 단계

표본 수집 단계는 딥 클러스터링을 위한 오토인코더 사전 학습용 이미지 자료를 수집하는 단계이다. 딥 클러스터링을 적용하기 위해서는 많은 양의 학습 데이터가 필요한데, 초기 크래시 발생 시 이미지 데이터가 없는 상황으로 가정하였을 때, 학습 자료를 수집하기 위해서는 가상 표본을 수집해야 한다. 소프트웨어와 비슷한 이미지 정보를 수집하여 학습할 수도 있지만, 소프트웨어 특성에 맞게 학습을 시켜야 정확한 분류가 될 수 있으므로, 소프트웨어를 실제 테스트한 이미지를 수집한다. 이미지 자료를 수집하기 위해서는 수동으로 스크린 캡처 정보를 추출하여 구성하는 방법이 있고, 소프트웨어 전체 과정을 테스트를 수행하여 n 초마다 스크린 캡처를 자동으로 수행하여 표본 이미지를 추출하는 방법이 있다. 이처럼 기본 동작 자료를 수집해서 초기에 크래시 정보가 없더라도 이미지 학습이 가능하며 소프트웨어가 주로 다루는 정보에 따라 해상도, 문자 표시 크기 등 성격에 따라 표본 데이터 크기를 효율적으로 미리 정할 수 있다. 본 연구에서는 n 초마다 스크린 캡처를 자동으로 수행하여 표본 이미지를 추출하는 방법으로 표본을 수집하고자 한다.

1.2. 학습 단계

학습 단계에서는 N2D 딥 클러스터링 기법을 적용하기 위하여, 오토인코더 신경망을 활용하여 표본 이미지를 학습시키고자 한다. 오토인코더는 결과물 해상도가 낮은 문제가 있으나, N2D 딥 클러스터링 기법에서의 오토인코더는 차원 축소를 위한 중간 데이터 추출 용도로만 사용하고 있어, 오토인코더를 최종 결과물을 추출하지 않는 다른 접근법을 사용하기 때문에 해상도는 식별할 수 없을 정도가 아니라면 크게 문제가 되지 않으며, 학습 시간을 절감시킬 수 있다. 오토인코더 네트워크 구성 내용은 Figure 9와 같다.

```
_________________________________________________________________
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
```

Figure 9. Resulting Parameters of Auto Encoder Neural Network

최적화 프로그램은 ‘Adam’을 적용하였으며, 모든 계층은 ‘ReLU’ 활성화 함수를 사용하였다. 중간 결과를 다음 알고리즘에 활용해야 하므로 인코더 마지막 단계에서 차원 수를 10개로 줄여 학습한다. epoch는 1000으로 설정하고, 최대 손실률을 0.05 미만으로 설정하여 학습을 종료하도록 하였다.

학습을 진행할 때에는 이미지 크기가 학습 속도 및 손실률을 결정하는 데 큰 비중을 차지한다. 학습 속도와 손실률을 고려하여 적절한 크기의 이미지 전처리

가 필요하다. 이미지 전처리 과정은 여러 가지 방법이 있으나, 본 논문에서는 이미지 축소만 적용한다. 이미지 축소가 필요한 크기는 소프트웨어 특징에 따라 다를 수 있다. 소프트웨어가 UI 위주로 동작한다면, 주요 UI 인터페이스를 식별할 수 있는 최소한의 크기까지 축소를 해야 하며, 게임과 같은 소프트웨어에서는 플레이어나 상호작용하는 객체들이 모두 식별 가능한 수준에서 축소를 진행해야 한다.

2. 자동 분류 설계

자동 분류 설계에서는 크래시 발생 직전 스크린 캡처 이미지와 발생 시 호출 스택 정보를 수집 방법과 수집된 정보를 가지고 분류를 진행하는 방법 및 순위화하는 과정에 대하여 세부적으로 설계한다.

2.1. 크래시 이미지 수집 방법

크래시 이미지를 자동으로 분류하기 위해서는 우선 크래시 정보를 수집해야 한다. 기본적인 크래시 정보는 Windows 기반 환경에서는 'dbghelp' 라이브러리를 활용하여 미니 덤프를 수집할 수 있다. 오류 관련 정보를 수집하기 위해서는 소스 코드에서 예외 핸들러를 등록하고, 예외 핸들러에서 덤프 및 호출 스택 정보, 스크린 캡처 이미지 정보를 저장하도록 하고, 크래시를 발생시킨다. 과정을 도식화하면 Figure 10과 같다. 미니 덤프 수집 시, 호출 스택 정보만 따로 텍스트 파일로 추출한다면 호출 스택 분류를 좀 더 수월하게 진행할 수 있다.

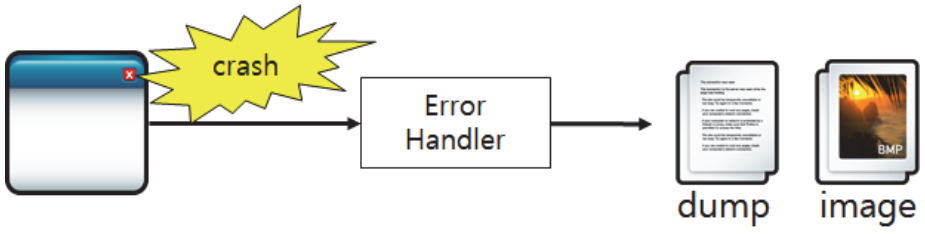

Figure 10. Crash Information Collection Process

크래시 발생 시점의 이미지를 추출하는 방법의 경우 여러 가지 방식이 존재한

다. 가장 범용적인 방법으로는 윈도 기반 API를 활용하여, 화면 DC를 얻어 스크린 캡처 이미지로 추출하는 방법이 있고, 다른 방법으로는 각 소프트웨어에서 활용하는 그래픽 라이브러리를 활용하는 방법이 있다. 윈도 기반 스크린 캡처로 대부분 이미지 추출이 가능하나, 그래픽 라이브러리를 사용한 경우에는 그래픽 라이브러리를 활용하여 화면을 캡처하면, 크래시 발생 시보다 더 정확한 화면을 캡처할 수 있다. 본 연구에서는 SDL 기반의 그래픽 라이브러리 기반 환경에서 이미지를 추출하였다. 이와 같은 방법으로 크래시를 확보하고 분류를 진행한다.

2.2. 호출 스택 분류

호출 스택을 기반으로 한 분류에는 2장에서 관련 연구에서 설명한 것과 같이 다양한 분류 방법이 있다. 본 논문에서는 호출 스택 기반 분류 시 발생하는 문제에 대하여 이미지 분류 기법을 적용한 결과를 확인하고자 최상위 호출 스택 내용을 기반으로 분류를 진행한다. 호출 스택 분류를 상세하게 할수록 이미지 분류도 더 명확하게 분류가 될 것으로 예상하지만, 호출 스택 분류를 상세하게 진행하는 것은 본 논문에서 얻고자 하는 결괏값이 아니므로 기본적인 분류만 진행하고, 상세하기 진행하지 않는다.

2.3. 이미지 분류

이미지 평가를 위해서는 N2D 딥 클러스터링 기법에 따라서 학습하여 저장한 압축 표현 층의 가중치 정보를 이용하여 1차 클러스터링 정보를 추출한다. 추출된 클러스터링 정보는 다시 UMAP 차원 축소 알고리즘을 적용한다. UMAP을 적용할 때, 하이퍼 파라미터값을 수정하여, 이웃의 개수는 20으로 설정, 순위화를 진행하기 위해 클러스터의 수를 차원의 수로 설정한다. 여기서는 10개의 순위화를 진행하기로 하여 클러스터의 수를 10으로 하였고, UMAP에서의 차원의 수도 10개로 설정한다. 또한, 점 사이 최소 거리를 0으로 설정한다. 점 사이 최소 거리를 작게 설정하면 차원 축소 구조가 좀 더 정확해지지만, 시각적으로는 확인이

불분명해질 수 있다. 여기서는 차원 축소에 관점을 두기 때문에 0으로 설정한다. 최종으로는 가우시안 혼합 모델(GMM) 알고리즘을 거쳐서 더욱더 명확하게 구분되도록 클러스터링을 진행한다. 가우시안 혼합 모델 알고리즘에서는 각각의 고유한 공분산 행렬이 있는 구성요소가 있다. 구성요소를 c라고 할 때, c는 클러스터의 수로 설정한다.

이렇게 딥 클러스터링 과정을 거치면, 10개의 순위화된 클러스터링을 거치게 된다. 여기까지의 과정을 보면 Figure 11과 같다.

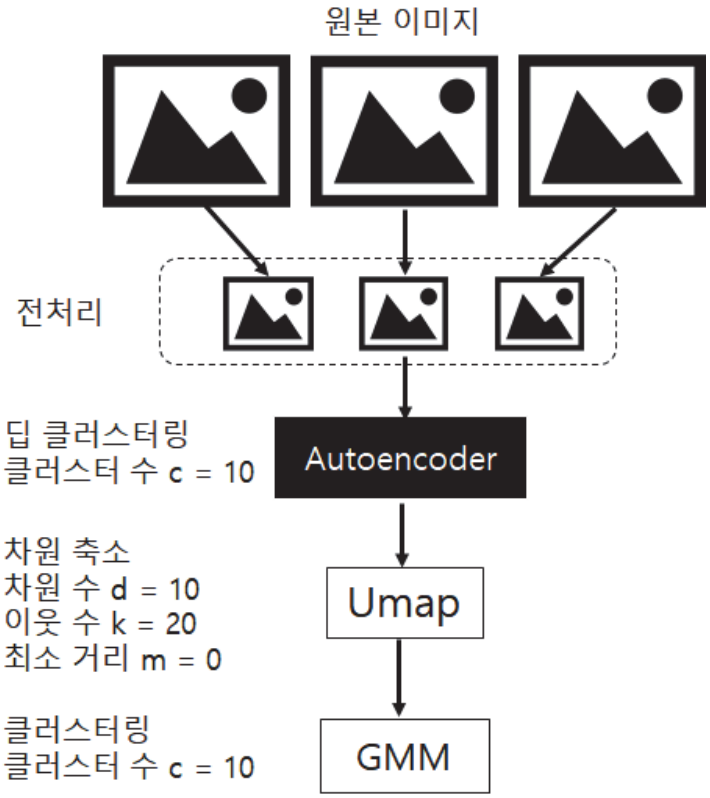

Figure 11. Image Deep Clustering Process

2.4. 순위화

마지막으로 클러스터링 결과를 토대로 순위화를 진행한다. 오토인코더, UMAP, GMM에서 클러스터링 수를 10으로 설정하였기 때문에, 10개의 분류가 생성되게 된다. 순위화를 적용하기 위해서는 다음과 같은 규칙에 따라서 순위화를 적용하

려고 한다.

첫째, 클러스터링 내 개체 수가 많은 그룹을 가장 높은 순위로 설정한다. 클러스터링 된 개체 수가 많다는 것은 크래시 발생 빈도가 높은 이미지라고 추측할 수 있다. 따라서 가장 많은 개체 수를 가진 그룹을 확인하면 가장 먼저 많이 발생하는 크래시에 대하여 대응할 수 있을 것이다.

둘째, 클러스터링 그룹별 개체 수가 같으면, 클러스터링 수치를 변경하여 개체 수를 비교해보고, 개체 수가 차이가 나는 클러스터링 수를 채택하도록 한다. 본 연구에서는 클러스터링 작업으로 고루 분포하는 결과보다는 지역성이 두드러지게 나타나는 결과가 더 중요하다. 순위별 차이가 명확해야 우선순위를 높여서 특정 그룹을 확인할 수 있으므로, 클러스터링 수를 변경해 보면서 추출해서 순위별 차이가 명확한 패턴을 찾도록 탐색을 할 수 있도록 한다.

클러스터링 수를 변경하기 위해서는 오토인코더, UMAP, GMM 단계별로 적용을 고려해야 한다. 오토인코더의 경우 초기 클러스터 수를 설정하여 학습하였기 때문에 변경하기가 어렵다. 따라서 클러스터 수의 변경은 그 이후 과정인 UMAP과 GMM 적용 단계에서 변경한다. 여기서 각각 UMAP의 차원 수 및 GMM의 클러스터 수를 Figure 12와 같이 2에서 10까지 변경하여 결과를 추출한다.

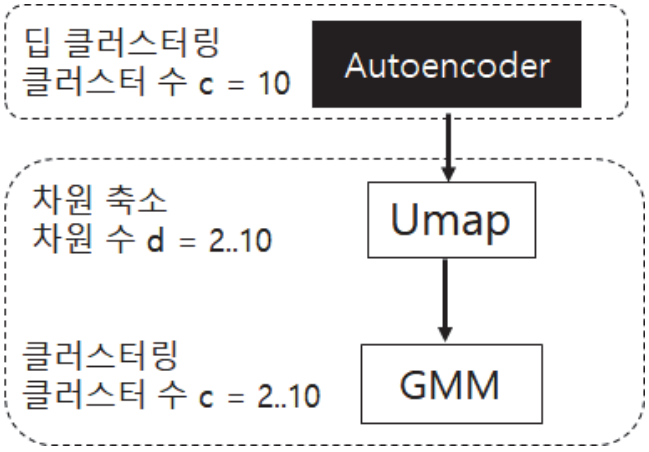

Figure 12. Change number of clusters for ranking

클러스터 수를 변경하여 추출하면 총 9개의 결과가 나오게 되는데, 각 그룹 별 개체 수에 대하여 각각 표준편차를 구하여 정보를 활용하여 계산하도록 한다. 표준편차는 다음과 같이 계산한다.

$$\sigma = \sqrt{\frac{\sum |x - \overline{x}|^2}{n}}$$

여기서 n은 클러스터 수와 같으며, x는 각 클러스터링 그룹별 개체 수이다. 각각의 클러스터 수별 표준편차를 구하고, 표준편차가 가장 큰 값을 채택하면 순위별 차이가 더 명확해질 수 있을 것이다.

위와 같은 방법으로 순위화를 진행한 다음 결과를 확인하여 이미지 순위 분류가 어떤 정보를 제공할 수 있는지 확인해본다.

IV. 구현 및 결과

1. 시스템 환경

본 연구의 실험 시스템 구현 환경은 Table 1과 같다. 표본 자료수집 및 크래시 정보 수집을 위해 크래시 덤프 발생 프로그램을 구현하였고, C++ 기반의 Visual Studio 2015 개발 IDE를 사용하였다. 본 연구에서는 이미지 분류를 보다 직관적으로 쉬운 예시로 확인하기 위해 이미지 확인이 명확한 단순한 게임을 선정하였고, ‘Super Mario Bros’ [12] 오픈 게임 소스를 활용하여 크래시 덤프 및 호출 스택 정보 및 스크린 캡처 이미지를 생성하였다. 게임 그래픽 라이브러리는 SDL을 사용하였고, 크래시 덤프 추출을 위하여 ‘dbghelp’ 라이브러리를 활용하였다.

학습 단계에서는 딥 클러스터링을 적용하기 위해 Python 3 기반의 Google Colaboratory 개발 IDE를 사용하였고, Tensorflow 및 Keras 라이브러리를 사용하여 알고리즘을 구현 및 테스트를 진행하였다.

Table 1. System environment information

운영체제 (OS)	Windows 10 64bit
CPU	Intel core i7-4790 CPU
Graphic Card	Nvidia Geforece GTX 960
크래시 덤프 개발 언어 및 환경	C++, Visual Studio 2015, SDL, dbghelp
딥 클러스터링 적용 개발 언어 및 환경	Python3, Google Colaboratory, Tensorflow 1.8.0, Keras

2. 구현 및 결과

2.1. 표본 자료수집 결과 분석

게임 기본 동작을 진행하고 3초마다 스크린 캡처 이미지를 생성하여 표본 이미지를 추출하였다. 이처럼 기본 동작 데이터를 통하여, 초기에 수집된 크래시 정보가 없더라도 학습할 수 있어 기반 시스템 구축이 가능하였다.

표본 이미지 수집은 'Super Mario Bros' 게임에서 총 4개의 단계를 1회 플레이하여 3초마다 스크린 캡처를 저장하여 총 388장의 이미지를 수집하였다. 수집된 정보의 단계 별 이미지의 예시는 Figure 13과 같다.

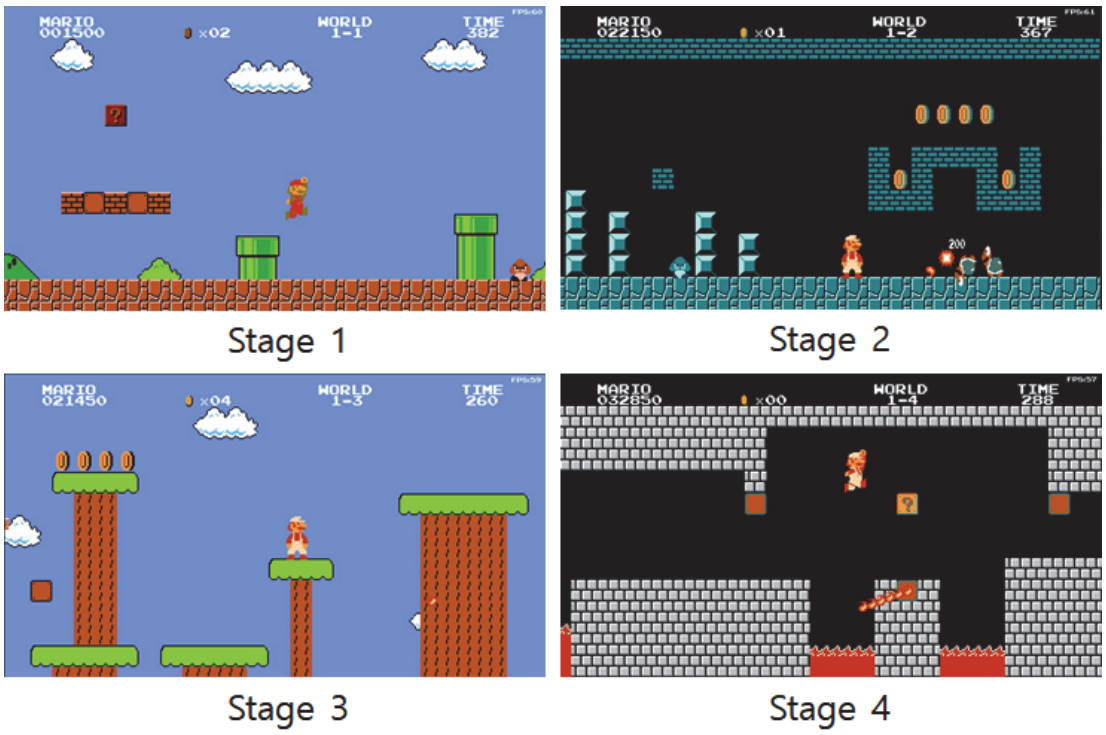

Figure 13. Sampled image per stage

단계별로 샘플링된 이미지를 살펴보면, 각 단계별로 배경 및 객체들의 차이가 있는 것을 확인할 수 있다. 또한, 단계 1과 단계 3은 배경이 같고 지형지물에서만 차이가 난다. 단계 2와 단계 4는 뒤 배경이 검은색이고 지형지물이 다르다. 클러스터링 결과에 대하여 예측해본다면, 1과 3은 클러스터링으로 구분이 어려운 편에 속할 것이며, 2와 4는 지형지물 등으로 구분이 가능할 것이다.

2.2. 오토인코더 학습

N2D 딥 클러스터링을 적용하기 위해 오토인코더를 학습을 진행하였고, 해당 알고리즘을 사용하여 적은 표본 이미지와 시간으로도 좋은 클러스터링 결과를 얻을 수 있었다.

1) 학습 단계 테스트

테스트를 위한 표본 이미지의 크기는 800 x 448 크기였지만, 방대한 크기로 인한 파라미터 증가로, 학습 시간이 기하급수적으로 늘어나는 문제가 있었다. 본 연구에서는 학습 시 메모리 부족이 발생하였고, 전처리로 절반 크기인 400 x 224 크기로 이미지 입력값을 축소하여 적용하였다. 압축 표현 층의 중간 결과를 활용해야 하므로 인코더(encoder) 네트워크 마지막 단계에서 차원 수를 클러스터 수인 10개로 줄여 학습하였다. 배치(batch)크기는 388로 설정하였고, 이미지 학습주기(epochs)를 100으로 설정하고 진행하였다. 초기에는 주기를 1000으로 설정하였으나 오히려 손실률이 즈가하는 상황이 발생하여 학습주기를 줄였고, 그 결과 손실률 0.04 정도로 만족할 만한 수준을 얻을 수 있었다. 학습된 가중치는 hdf5 포맷으로 저장하였다.

2) 학습 테스트 결과 분석

학습 데이터를 테스트하는 과정은 3장에서 설명한 이미지 분류 단계와 같다. 학습된 이미지를 그대로 클러스터링한 결과를 추출하고, 이미지가 잘 분류되었는지 눈으로 확인한다. 분류를 진행하는 과정을 요약하면 Figure 14와 같다. 우선 이미지를 전처리하여 400 x 224 크기로 축소한 뒤, 오토인코더 압축 표현 층의 가중치 정보를 가지고, 1차 클러스터링을 진행한다. 그 후 UMAP 차원 축소 및 GMM을 적용하여 최종적으로 분류된 이미지를 확인한다.

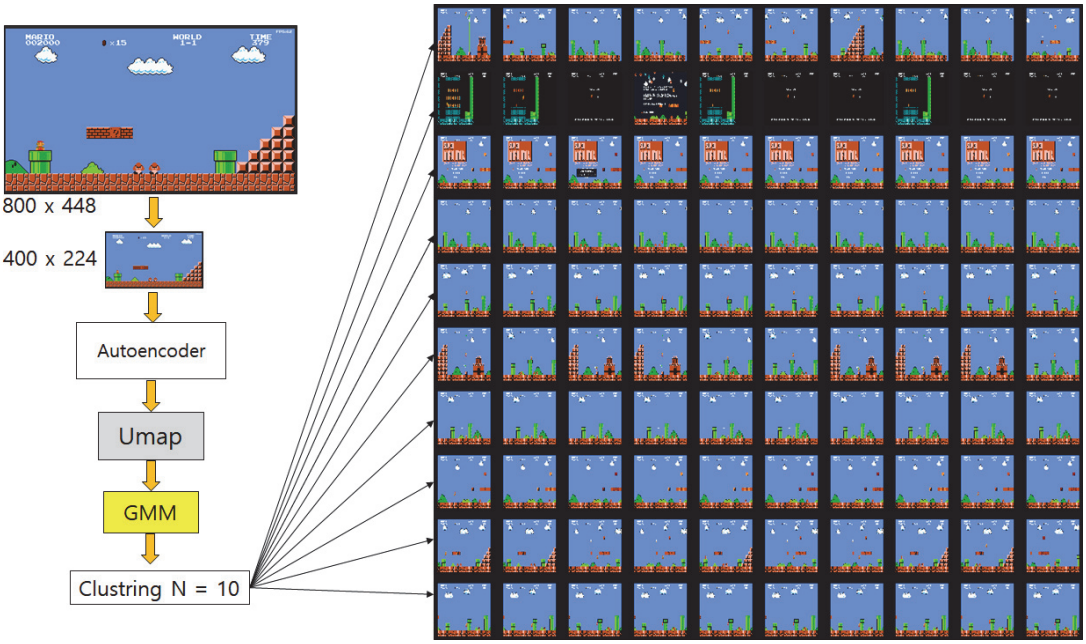

Figure 14. The process of classifying clustering based on the sampled image

학습된 가중치로 클러스터링이 잘 되는지 테스트를 진행한 결과는 Figure 15, Figure 16과 같다.

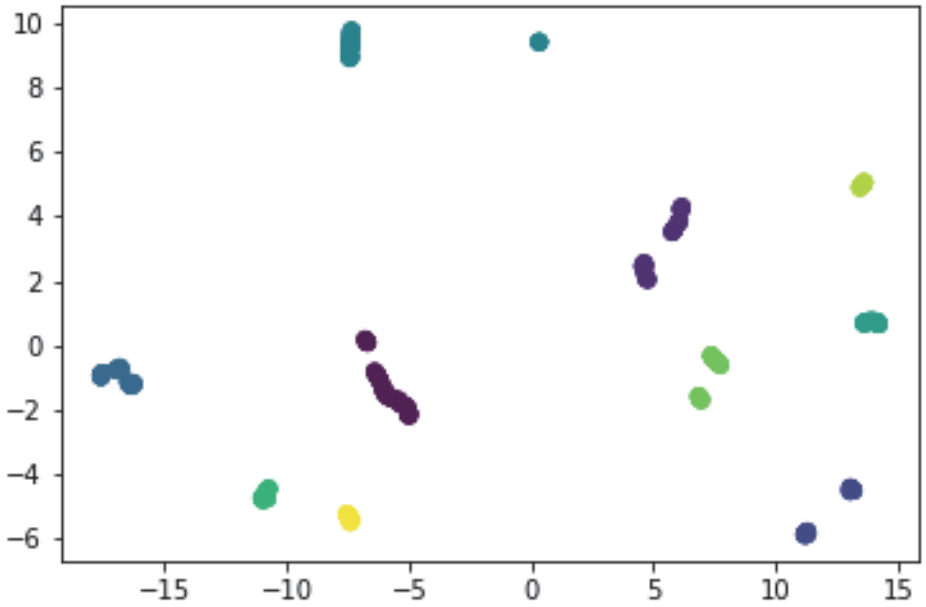

Figure 15. Clustering Results Visualization

클러스터링 시각화를 위해, Figure 15와 같이 2차원 공간에서의 결과를 표시하였고, 클러스터별로 다른 색상으로 표시하였다. 결과를 보았을 때, 각 클러스터가 2차원 공간에서 겹치지 않게 고루 분포한 것을 볼 수 있다.

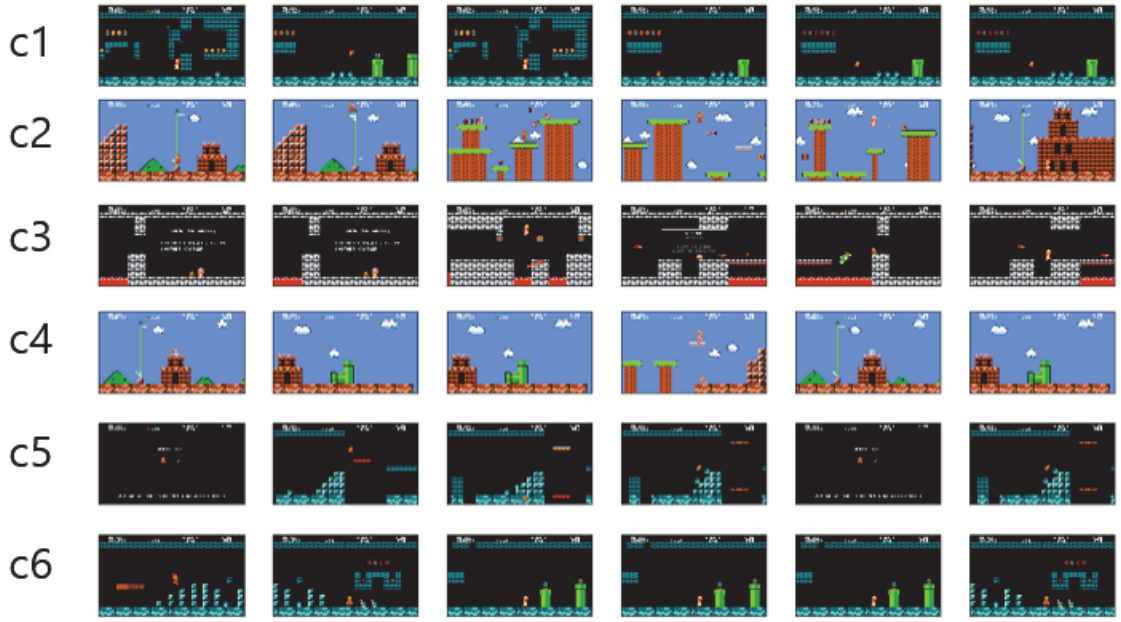

Figure 16. Some of the play data based clustering results (showing 6 of 10)

Figure 16에서 클러스터링 결과 그룹 C1 ~ C6를 확인하였을 때, {C1, C3, C5, C6}는 단계 구분이 비교적 명확 하였지만 {C2, C4}의 경우는 두 개의 단계가 혼합된 결과를 보였다. 이는 2.1절 분석에서 예측한 것과 같이 배경 이미지가 비슷한 것으로 인한 결과로 보인다. {C1, C6}, {C2, C4}의 경우에도 서로 비슷하여 두 그룹의 차이를 비교해보았고, {C1, C6} 그룹에서는 모두 단계 2 이미지로 구성되어 있는데, 단계별 특징에 따라서 이미지가 분류된 것을 볼 수 있다. {C2, C4} 그룹에 대해서는 C4는 구간이 단조로운 데 비해서 C2는 지형 정보가 더 많아서 복잡해 보여서 따로 분류되었음을 추측할 수 있었다. 위 결과로 보았을 때, 어느 정도 분석이 가능할 정도로 의미 있게 분류가 되었음을 알 수 있다.

2.3. 크래시 덤프 발생 및 결과 분석

크래시 이미지를 분류하기 위해서는 우선 크래시를 발생시켜야 한다. 크래시를 강제로 발생시키기 위해서 ‘Super Mario Bros’ 충돌 체크 코드에서 캐릭터와 적 객체 간의 접촉 시 간헐적으로 크래시를 발생시키는 코드를 Table 2와 같이 작성하였다.

Table 2. Code providing the cause of the crash

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

크래시를 간헐적으로 발생시키기 위해 랜덤 함수를 사용하여 50% 확률로 크래시 코드를 적용하게 하였다. 이렇게 적용하는 이유는 다양한 구간에서 크래시가 발생하도록 하기 위해서이다. Table 2와 같이 작성한 코드는 해당 충돌 체크 함수에서는 문제가 발생하지 않으나, ‘nullptr’을 대입한 객체를 참조하는 시점에서 널 포인터 체크를 하지 않으면, 크래시가 발생한다. 이와 같은 크래시 유형은 객체를 참조하는 시점과 잘못된 소스 코드가 있는 위치가 서로 일치하지 않기 때문에 호출 스택 기반 분류로 확인하였을 때는 최상위 호출 스택이 다르게 보일 가능성이 매우 크다. 최상이 호출 스택이 다르게 보인다는 것은 2장에서 제시한 분류 문제점 중에서 서로 같은 유형을 다른 유형으로 분류하는 문제에 속한다. 또한, 이러한 널 포인터 참조 오류는 원인이 발생 코드 위치를 찾기가 어려워 객체가 널 포인트가 된 지점을 직접 디버깅해보거나 재현을 해보지 않으면,

원인을 파악하기 어려운 상황이 발생할 가능성이 크다.

Table 2의 내용을 추가한 오류 코드가 삽입된 프로그램을 테스트해 총 58건의 크래시를 발생시켰고, 그 결과 호출 스택을 분류하였더니, Table 3과 같이 두 종류의 크래시 유형 분류가 발생하였다.

Table 3. Crash call stack type derived from Table 2

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

A 유형은 58건 중 47건으로 대부분의 크래시가 이 유형으로 분류되었고, 최상위 호출 스택에 있는 함수가 Minion::getBlockID()이다. 호출 스택 내용만으로는 화면을 그리면서 적을 그리다가 블록 객체를 참조하다 죽은 것처럼 보인다. Table 2에서 삽입한 충돌이 발생하는 함수 'Map::UpdateMinionsCollisions()'에 대한 내용은 호출 스택에서 찾아볼 수 없다.

B 유형은 58건 중 17건으로, 최상위 호출 스택 함수는 'Player::getXPos()'이다. 여기서는 플레이어 객체를 참고하다 크래시가 발생하였음을 예측할 수 있으며, 'Map::UpdateMinionsCollisions()' 함수가 다음 스택에 있어서 원인을 제공하는 코드를 쉽게 찾을 수 있다.

위와 같이 A 유형과 B 유형이 발생하는 위치는 같으나, 두 유형의 충돌이 하나의 함수로부터 원인이 발생하여 서로 연관되어 있을 것으로 추측하기 힘들다. 만약 개발자가 Table 2의 내용을 미리 알고 있다면, A 유형은 'Minion[i][j] = nullptr'이 원인이고, B 유형은 'Player = nullptr'이 원인임을 쉽게 유추할 수

있지만, 크래시 발생 원인을 모르는 상태에서 이를 유추하기는 쉽지 않다. 예를 들어 A 유형의 경우만 확인할 경우, 호출 스택 정보만으로는 어느 과정에서 원인이 발생하였는지 유추하기 어렵다. 따라서 호출 스택 정보만으로 A와 B 유형의 공통점을 찾을 수 없기 때문에 같은 위치에서 파생했을 것이라는 유추는 기존 호출 스택 기반연구에서는 확인하기 어렵다. 이러한 부분에서 이미지 기반 분류를 적용하였을 때, 다음 절에서 결과를 분석해보기로 한다.

2.4. 이미지 기반 분류 결과 분석

크래시 유형별로 분류 방법 및 유형 구분 없이 분류하는 방법으로 나누어 분류를 진행하고 분석하였다.

2.4.1. 유형별 이미지 결과 분석

1) A 유형 이미지 분류 결과 분석

호출 스택 기반으로 분류된 이미지를 이미지 클러스터링을 통하여 추가 분류한 결과로 A 유형의 결과는 Figure 17과 같다.

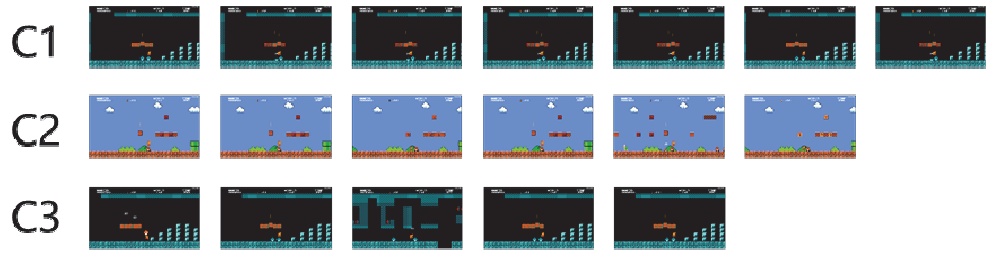

Figure 17. A type secondary image classification result
(the top three of c = 10 are displayed)

A 유형 이미지 분류에서 주요 순위 3개만 확인하였을 때, C1이 7건, C2가 6건, C3가 5건으로 A 유형 전체 개수 중의 38%가 분류되었다.

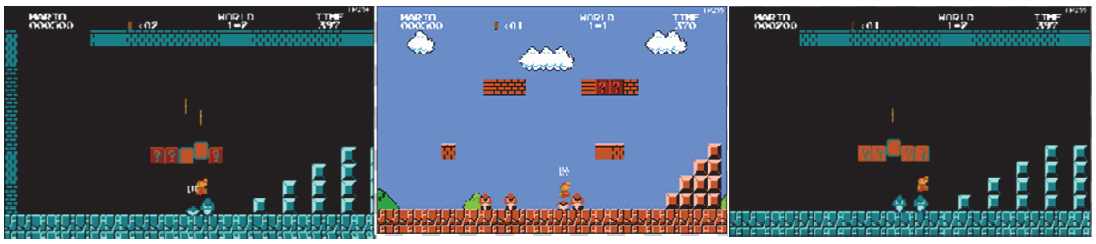

Figure 18. Analysis of commonalities of representative images

각 순위의 대표 이미지들은 Figure 18과 같다. 높은 순위의 이미지들의 공통점을 분석해보면 다음과 같다. 첫째, 캐릭터와 적과의 접촉이 있다. 둘째, 적 객체가 둘 이상이다. 셋째, 캐릭터는 공중으로 점프한 상태이다. 이처럼 이미지에서 공통적인 부분을 확인함으로써 원인에 대한 많은 힌트를 얻을 수 있었다. 또한, Figure 17의 {C1, C2, C3}의 경우, 같은 단계의 같은 구간 이미지가 많은 것으로 보아서 해당 위치에서 크래시 될 확률이 높다는 것도 유추할 수 있었다.

따라서, 이미지 분류만으로도 호출 스택에서는 찾기 어려운 부족한 주변 상황 정보들을 상당히 많은 얻을 수 있음을 알 수 있다.

2) B 유형 이미지 분류 결과 분석

B 유형의 경우 추출된 결과물이 총 17건으로 클러스터 수를 10개로 설정하고 분류하였을 때 결과는 Figure 19와 같다.

Figure 19에서 볼 수 있듯이 클러스터당 1~3건의 이미지가 고루 분포되어 있어서 순위는 C1, {C2, C3, C4, C5, C6}, {C7, C8, C9, C10}으로 순위별로 동일 순위 그룹이 존재한다. 이러한 결과는 우선순위 분류에 적절하지 않고, 전체적으로 보았을 때도 같은 이미지가 매우 많이 보이는 것을 알 수 있다. 따라서 크래시 수량에 따라 적절한 클러스터 수를 조정해야 한다.

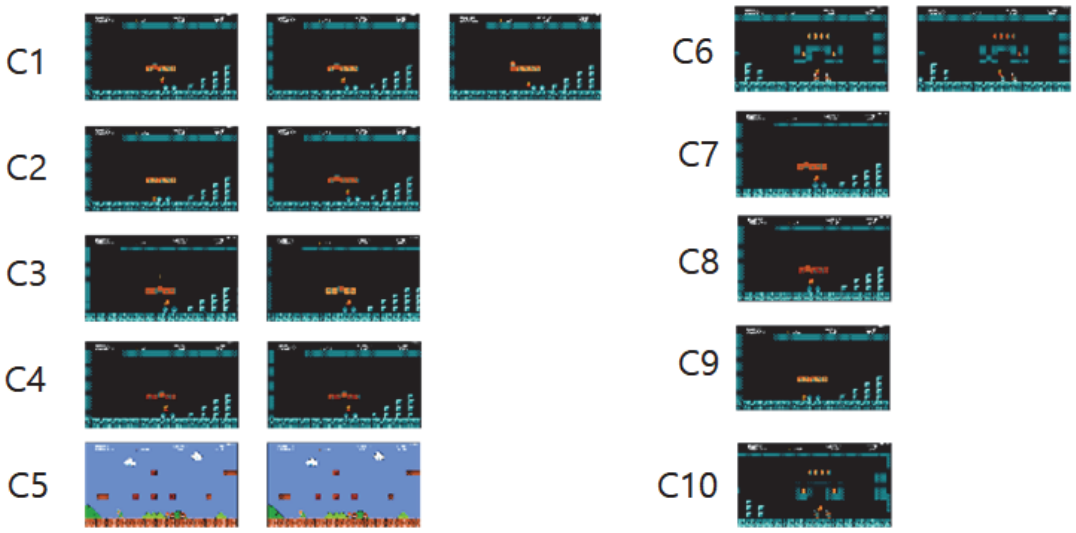

Figure 19. Classification results when the number of clusters in type B is 10

적절한 클러스터 수를 자동으로 선정하여 추출하기 위해 각 클러스터 수에 따른 그룹별 표준편차를 구해 보았다. c 값을 변경하여 다시 분류를 진행하였을 때 표준편차 결과는 Table 4와 같다.

Table 4. Standard Deviation from Changing the Number of Clusters

그룹	A	B	C	D	E	F	G	H	I	표준 편차
C=10	1	2	2	2	3	1	2	1	1	0.707106781
C=5	4	5	3	2	3					1.140175425
C=3	7	5	5							1.154700538

c가 3, 5, 10일 경우의 각각의 표준편차를 구해 보면, c가 3일 경우에 표준편차가 가장 높은 것으로 나타났다. c가 3, 5, 10일 때의 그룹별 이미지를 확인하면 Figure 20과 같이 분류된다.

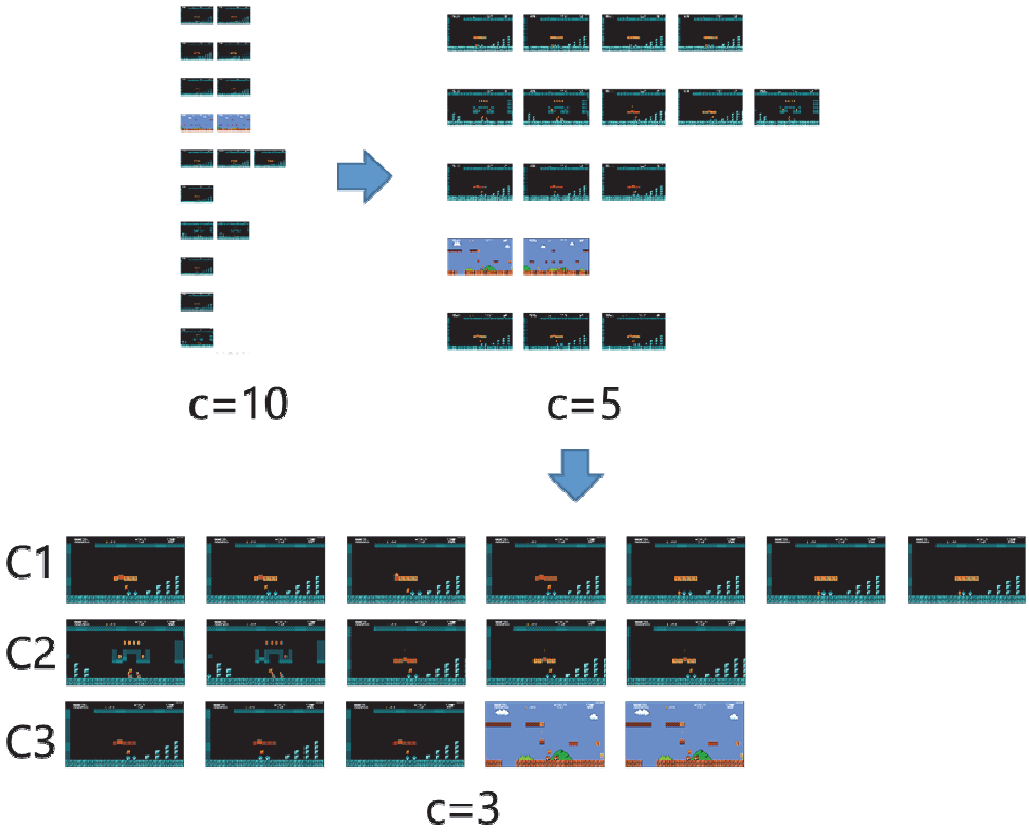

Figure 20. Result after changing the number of clusters of type B

클러스터 수를 10개로 하였을 때는, 그룹당 2개씩 고루 분포하였던 결과가 클러스터 수를 3개로 줄였을 때는 {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 주원인이 있음을 가정하고 버그를 추적해보는 것이 더욱 효율적일 것이라는 결론을 이미지 분류를 통해 유추할 수 있다. 또한, 재현 및 해결을 확인하기 위한 테스트를 진행하려면 A, B 유형 공통 1순위인 두 번째 단계에서 테스트를 진행해 보는 것이 유리하다는 것을 알 수 있다.

2.4.1. 유형 구분 없는 이미지 분류

호출 스택 유형 구분 없이 이미지 분류를 하였을 때, 어떤 결과가 나오는지 확인하고자 A 유형과 B 유형을 혼합한 58개의 유형 구분 없는 이미지를 가지고 분류를 진행하였다. 결과는 Figure 22와 같다. 가장 많은 요소를 가지고 있는 그룹은 C0, C2, C3이고, Figure 13에서의 단계별 대표 이미지를 참고하면, 단계 2의 이미지가 절반 이상을 차지하는 것을 알 수 있다. 또한, C0, C2, C3, C6, C8 그룹에서 상당히 유사한 이미지들이 반복되는 것을 확인할 수 있다. 이미지가 어떤 속성으로 이렇게 분리가 되었는지는 알기가 어려우므로, UMAP 알고리즘을 적용한 시점의 결과를 시각화하여 분석하려고 한다. 시각화한 결과는 Figure 21과 같다.

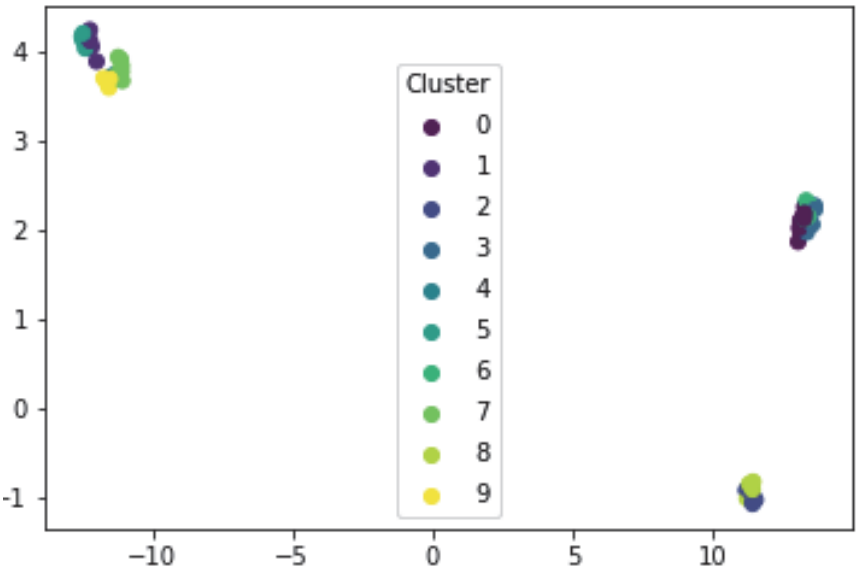

Figure 21. Visualization result when the number of clusters is set to 10

클러스터 수를 기본적으로 10개로 설정하였으나, 2차원으로 시각화한 공간에서는 크게 3개의 영역으로 주로 나누어져 분포하고 있는 것으로 나타났다. Figure 22와 비교하여 그룹을 살펴보면, 왼쪽 위 A 영역은 {C1, C4, C5, C7, C9}, 오른쪽 가운데 B 영역은 {C0, C3, C6}, 오른쪽 아래 C 영역은 {C2, C8}으로 특징 분

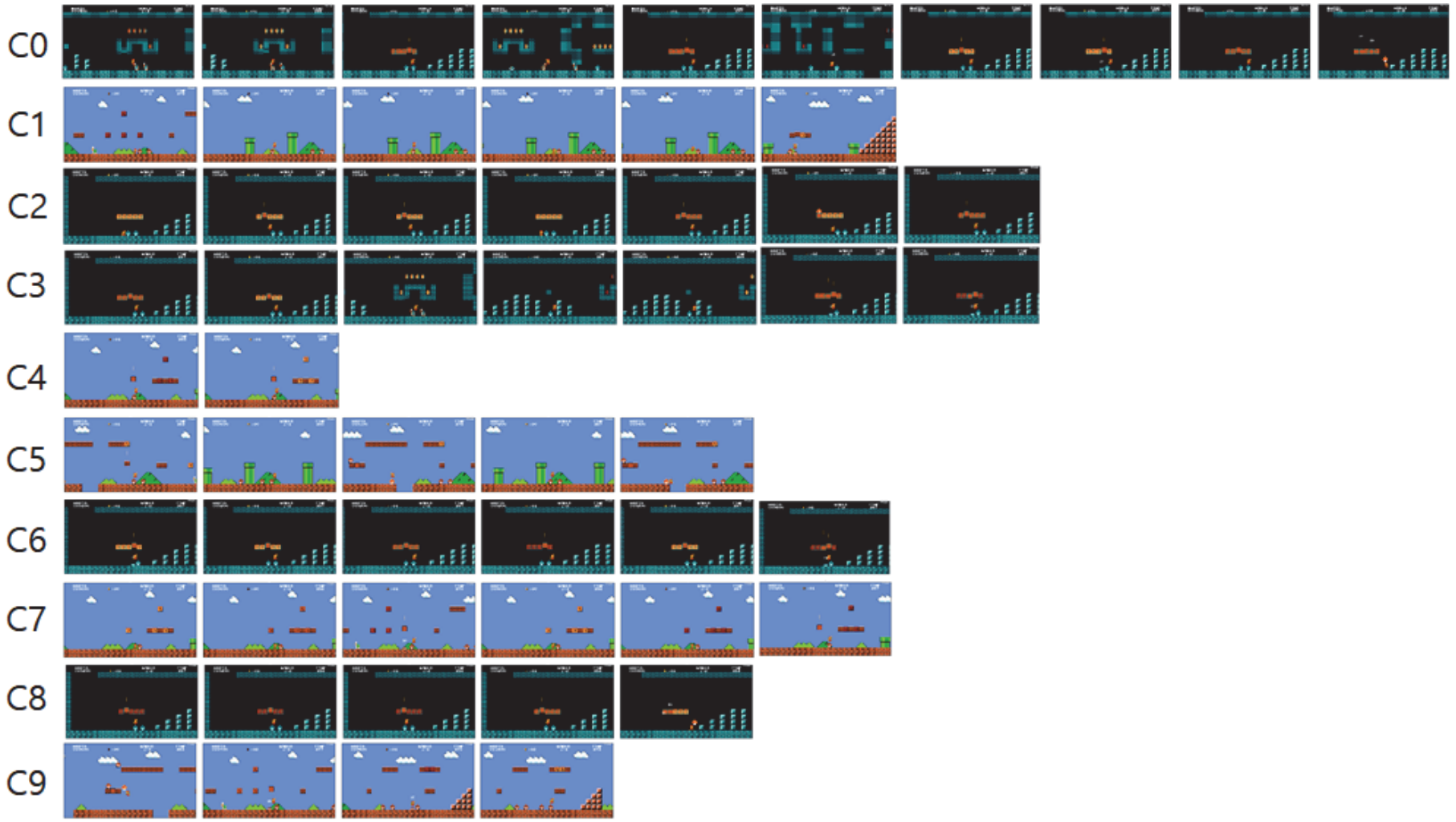

Figure 22. Clustering result without type distinction

Table 5. Key figures change as the number of clusters decrease

총 클러스터 수	클러스터별 요소 분포 개수										평균	중앙값	표준편차
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9			
10	10	6	7	7	2	5	6	6	5	4	3.93	3	2.97
9	10	6	7	7	6	5	6	6	5	-	3.59	3	2.63
8	10	6	12	7	6	5	6	6	-	-	3.07	3	2.28
7	4	5	12	13	12	6	6	-	-	-	3.14	3	1.64
6	10	5	12	13	12	6	-	-	-	-	2.51	3	1.58
5	10	11	12	13	12	-	-	-	-	-	2.1	2	1.39
4	10	23	12	13	-	-	-	-	-	-	1.48	1	1.02
3	23	23	12	-	-	-	-	-	-	-	0.81	1	0.75
2	35	23	-	-	-	-	-	-	-	-	0.39	0	0.48

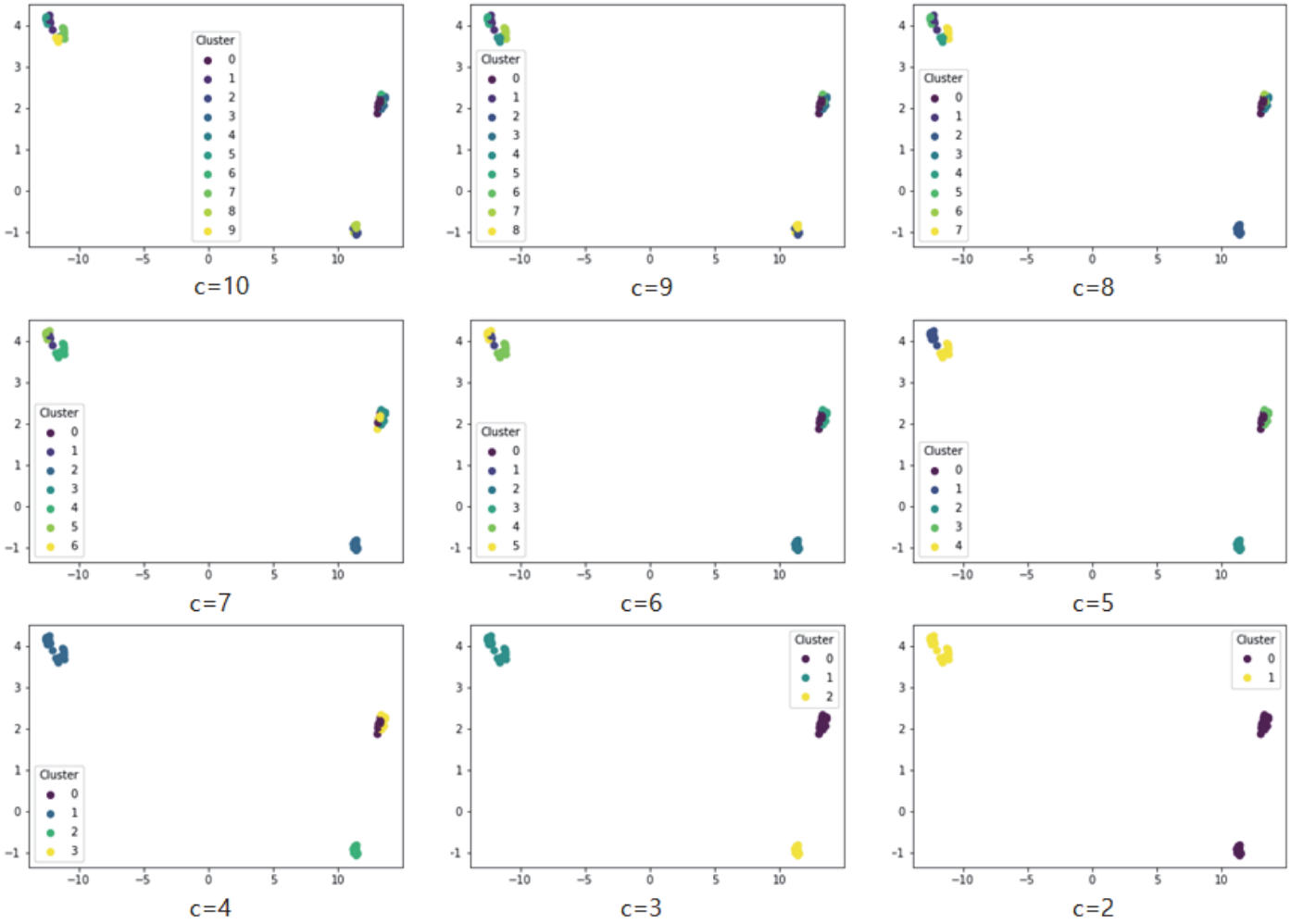

Figure 23. Clustering visualization results without type distinction

포가 나눠질 수 있을 것으로 보인다. 또한, 2차원으로 시각화한 분류 공간을 분석해보면 A 영역과 B, C 영역 사이의 거리가 상대적으로 더 떨어져 있음을 알 수 있다. Figure 22를 보면 A 그룹에 해당하는 클러스터들은 1단계 이미지에 해당하고, B, C의 이미지는 2단계에 해당하는 것을 알 수 있다. 따라서 영역 정보가 단계별 특성을 잘 반영하고 있음을 알 수 있다.

Table 5는 총 클러스터 수를 10개에서 2개까지 줄였을 때의 수치를 기록한 결과이다. 2.4.1 절에서 이미지 분류 시 표준편차를 이용하여 표준편차가 큰 클러스터를 채택하여 분리하였는데, Table 5의 결과에서는 표준편찻값이 점점 감소하는 수치를 보였다. 따라서 값으로 보았을 때는 어떤 결과가 적절한 수준인지 결정하기 어려웠다.

Figure 23은 총 클러스터 수를 10개에서 2개까지 줄였을 때의 과정을 2차원으로 시각화한 결과를 출력한 그림이다. 총 클러스터 개수가 감소할수록 영역별로 그룹화가 잘 통합되고 있음을 확인할 수 있다. 크게 3개의 A, B, C 특정 영역으로 나누어졌다고 할 때, 처음에는 각 영역에 여러 개의 클러스터가 분포하게 되는데, 총 클러스터를 줄이면서 각 특징 그룹이 하나의 클러스터로 이루어지는 시점을 분석하면 다음 Table 6과 같다.

Table 6. Cluster Integration Points by Visualization Area

영역	클러스터 통합 완료 시점	클러스터 통합 과정
A 영역	c=4	c=10 : {C1, C4, C5, C7, C9} c=9 : {C1, C4, C5, C7} c=8 : {C1, C4, C5} c=5 : {C1, C4} c=4 : {C1}
B 영역	c=3	c=10 : {C0, C3, C6} c=5 : {C0, C3} c=3 : {C0}
C 영역	c=8	c=10 : {C2, C8} c=8 : {C2}

A, B, C 영역은 클러스터 수가 8개, 4개, 3개로 줄어들면서 점차 통합되었다. 특히 C 영역의 경우에는 클러스터 통합 시점이 다른 영역보다 매우 빠르게 나타났

다. C 영역에서 통합된 이미지 결과는 Figure 24와 같다. C 영역의 경우 A, B 통합되었을 때의 이미지와 비교하였을 때 가장 유사한 이미지들이 통합된 것을 볼 수 있었다. 따라서 특징이 명확한 그룹일수록 통합이 빨리 될 수 있을 것으로 추측할 수 있다. 순위화 분류를 진행할 때, 특정 영역의 통합시기를 할 수 있으면 보다 효과적으로 클러스터링 결과를 반영할 수 있을 것으로 보인다.

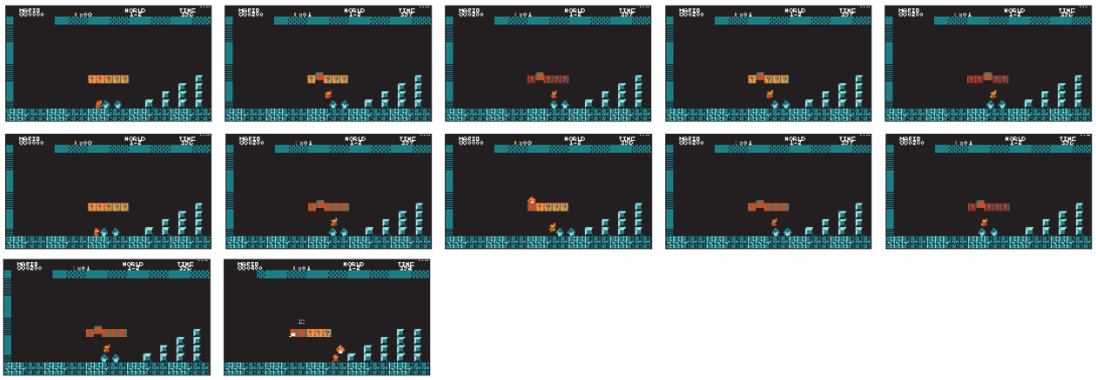

Figure 24. Image of C area cluster consolidation point

Table 6의 A 영역의 경우에는 상대적으로 많은 클러스터 그룹이 있었고, A 영역의 이미지들이 1단계 이미지로만 구성되어 있다는 것으로 보아 크래시 위치가 다양하다는 것을 추측할 수 있다. Table 6의 클러스터 통합 과정을 참고하여 크래시 재현 테스트를 위한 정보를 얻을 수 있을 것이다. 재현 테스트는 문제 발생 원인을 찾지 못하였을 때, 직접 같은 크래시를 발생시켜 디버깅할 수 있으므로 중요하다. Table 6에서 보았을 때, 한 영역 내에 클러스터 그룹의 개수가 적으면서, 통합이 빨리 이루어지는 순서대로 보았을 때, C, B, A 순서로 접근하는 것이 도움이 될 것으로 보인다.

V. 결론

본 연구에서는 호출 스택 분류 기반의 기존 연구들에서 크래시가 발생하는 시점과 잘못된 소스 코드가 있는 위치가 서로 일치하지 않는 경우 호출 스택이 전혀 다르게 나오기 때문에 호출 스택 기반 분류가 어렵다는 한계점을 확인하였다.

이를 개선하기 위해서 크래시 이미지를 활용하는 방법을 제시하였다. 크래시 이미지 수집하기 전에 딥러닝 모델 학습을 위한 표본 이미지를 추출하는 방법에 대해 제시하였고, 오토인코더 딥 클러스터링 모델과 UMAP과 GMM 알고리즘을 활용한 N2D 모델을 활용하여 이미지 유사성에 따라 자동으로 분류하는 클러스터링 모델을 구현하였고, 자동 분류 결과를 크래시 해결을 위해 중요하다고 판단되는 순서로 순위화 방법을 제시하였다. 또한, 호출 스택 정보와 함께 이미지 정보를 활용하여 분류하는 방법과 이미지 정보만으로 분류하는 방법에 대해 분석해 보았고, 분류하는 방법에 따라서 크래시 해결을 위한 다양한 정보를 제공할 수 있다는 것을 실험을 통해 알 수 있었다.

호출 스택 정보와 함께 이미지 분류를 하면 호출 스택 분류의 한계점을 보완하여 크래시 원인을 발생하는 소스 코드 위치를 알 수 없는 경우에도 일관된 크래시 이미지 결과를 통하여 원인이 발생한 지점에 대하여 유추할 수 있었고, 이미지 분류 정보만을 활용하면 어느 구간에서 크래시가 많이 발생하는지를 직관적으로 확인할 수 있으며, 재현을 위한 테스트 지점 설정에 큰 도움을 줄 수 있음을 알 수 있었다.

크래시 이미지를 순위화하기 위해서는 기본적으로는 유사성 높은 이미지들이 많은 그룹을 높은 순위를 매기는 방식이 유효하나, 효과적인 크래시 이미지 유형을 분류하려면 다양한 접근 방법이 필요하다는 것을 알 수 있었다. 최대 분류 순위를 클러스터 수로 결정하였는데, 클러스터 수를 줄여감에 따라 다양한 분류 결과가 나왔으며, 특징이 명확한 그룹일수록 통합이 잘되는 것도 확인할 수 있었다. 이미지 분류 방법을 결정할 때, 클러스터링 이미지 분포를 시각화하면 좀 더 의미 있는 정보를 찾을 수 있음도 시각화 분석을 통하여 알 수 있었다. 우선순위

설정은 재현 테스트가 중요한지, 다양한 상황별 분석이 필요한지 등 사용자가 어떤 관점에서 크래시 이미지를 활용하는지에 따라서 순위가 결정될 수 있으므로, 목적에 따라서 다양한 방법으로 순위화를 진행할 수 있음을 본 연구를 통하여 알 수 있었다. 또한, 크래시 이미지 기반 결과 분석은 코드 분석보다 어렵지 않기 때문에 비 개발자들도 충분히 활용할 수 있다는 점도 소프트웨어 크래시 해결을 위한 품질 개선의 의미가 있다.

앞으로는 크래시 이전에 연속적인 이미지 및 음향 등의 시계열 정보를 활용한 크래시 분류 연구나 클러스터링 된 정보를 좀 더 의미 있는 특징이 나타나도록 시각화 도구를 개발하거나 시각화된 결과를 스스로 분석하여 순위화를 자동으로 분류하는 기법 등의 연구가 필요하다.

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

[5] 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

[6] 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.

[7] 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.

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

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

[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