논문 21-46-02-24 The Journal of Korean Institute of Communications and Information Sciences '21-02 Vol.46 No.02
https://doi.org/10.7840/kics.2021.46.2.390

실시간 분석을 위한 도커 컨테이너 기반의 딥러닝 모델 관리 시스템 설계 및 성능 비교

이 모 세◆, 강 민 수*, 김 인 호*, 김 재 헌°

Design and Performance Comparison of Docker Container Based Deep Learning Model Management System for Real-Time Analysis

Mo-se Lee,◆ Min-su Kang*, In-ho Kim*, Jae-hun Kim°

요 약

딥러닝 기술은 고차원의 데이터에 대한 결과를 추론하는데 매우 효과적이어서 다양한 비즈니스 영역에 적용할 수 있는 것으로 알려져 있다. 딥러닝 프레임워크의 발전으로 다양한 인공지능 서비스들이 등장하고 있다. 특히 최근 화두가 된 스마트시티는 도시에서 발생하는 다양한 데이터들을 기반으로 한 인공지능 서비스를 제공한다. 그러나 대부분의 인공지능 서비스들은 딥러닝 기술을 이용한 모델의 추론 정확도를 만족시킬 뿐 리소스를 효율적으로 사용하는데 한계점이 있다. 실제 서비스 적용을 위해 웹 또는 플랫폼 아키텍처를 구성할 때에도 데이터 유입부터 전처리, 모델 학습, 서빙에 대한 다양한 인프라 기술을 운영하기 위한 비용이 발생된다. 스마트시티 서비스의 경우 방대한 데이터를 기반으로 운영되기 때문에 운영비용이 크게 발생할 것으로 보인다. 본 연구에서는 딥러닝 모델을 적용하는 안정적인 웹서비스 개발과 배포 및 관리를 용이하게 하기 위해 다양한 아키텍처를 설계하고 아키텍처들 간의 성능을 비교하고자 한다. 일반적인 함수를 실행하는 방법인 Embedded Function 아키텍처와 REST 방식인 Flask API, Fast API, TFServing 아키텍처들 총 4가지에 대해 성능 비교를 진행하였다. 실험 결과 데이터 처리 속도 면에서는 TFServing이 가장 뛰어난 성능을 보여주었지만 탑재 가능한 모델의 종류, 커스터마이징 면을 고려했을 때에는 충분한 속도를 보여주는 Fast API도 안정적인 아키텍처로 개발하기에 충분하다는 결과를 보여주었다.

키워드 : 딥러닝, 딥러닝 모델 관리, 실시간 분석, 서빙 아키텍처, 도커 컨테이너
Key Words : Deep Learning, Deep Learning Model Management, Real-time Analysis, Serving Architecture, Docker Container

ABSTRACT

Deep learning technology is known to be very effective in inferring the results of high-dimensional data and can be applied to various business areas. With the development of the deep learning framework, various artificial intelligence services are emerging. In particular, Smart City, which has recently become a hot topic, provides artificial intelligence services based on various data from cities. However, most artificial intelligence services have limitations in efficiently using resources only to satisfy the inference accuracy of models using

※ 본 연구는 스마트시티 혁신성장동력프로젝트(1615011527) 과제의 연구비 지원에 의해 수행되었습니다.
◆ First Author : VReducation Co.,Ltd, lms7140@vreducation.kr, 책임연구원, 정회원
° Corresponding Author : VReducation Co.,Ltd, sean@vreducation.kr, 대표이사, 정회원
* VReducation Co.,Ltd, kmsjks79@vreducation.kr, 선임연구원, 정회원; JC Square Inc., otiger@jc-square.com, 부장, 정회원
논문번호 : 202011-272-0-SE, Received October 28, 2020, Received December 1, 2020, Accepted December 1, 2020

deep learning technology. Even when configuring a web or platform architecture for actual service application, costs for operating various infrastructure technologies from data inflow, pre-processing, model learning, and serving are incurred. In the case of smart city services, operating costs are expected to be large because they are operated based on vast amounts of data. In this study, various architectures are designed for the development, distribution, and management of stable web services equipped with deep learning models, and the performance of the architectures is compared. Performance comparisons were made for a total of four architectures: Embedded function architecture, which is a general function execution method, and Flask API, Fast API, and TFServing architectures, which are REST methods. As a result of the experiment, TFServing showed the best performance in terms of data processing speed, but when considering the types of models and customization that can be mounted, Fast API, which shows sufficient speed, was also shown to be sufficient to develop a stable architecture.

Ⅰ. 서 론

딥러닝(Deep Learning)은 활발하게 연구되고 있는 분야로써 다양한 종류의 신경망 계층을 조합하여 복잡한 자료들 속에서 핵심적인 내용을 요약하는 기계학습(Machine Learning)의 한 분야이다[1]. 딥러닝은 인공신경망(ANN, Artificial Neural Network)에 기반하여 설계된 개념이다. 인공신경망은 기계학습과 인지과학에서 동물의 뇌신경망을 착안하여 만들어진 알고리즘으로 1940년 후반부터 연구가 시작되었다. 당시 높은 성능을 보여주던 SVM(Support Vector Machine) 기법보다 더 높은 정확도를 보여주었지만, 인공신경망의 방대한 연산량으로 인해 거의 사용되지 않았었다[2]. 그럼에도 인공신경망에 대한 연구는 지속되었으며, 연구자들은 1980년대에 기존 신경망의 계층수를 증가시키는 심층신경망(DNN, Deep Neural Network)의 개념을 제안했다. 기존 신경망에 비해 더 많은 계층을 활용하여 복잡한 데이터들에 대한 표현능력과 성능을 크게 향상시킬 수 있었다[3]. 그러나 당시 하드웨어의 성능이 심층신경망을 구축하기에는 부족하였다.

이러한 심층신경망의 아이디어는 Hinton 교수에 의해 다시 주목받게 되었고, 2006년 논문에서 사전 학습(Pre-training)이라는 개념을 제안하여 심층신경망의 학습 가능성을 보여주었다[4]. 사전 학습은 신경망의 초기 값을 임의의 값에서 시작하는 것이 아니라, 사전 학습을 통해 심층신경망의 연결을 학습에 도움이 되는 값으로 미리 변형하는 것을 말한다. 이를 통해 효과적인 심층신경망의 학습이 가능해졌고 이후 연구가 활발히 일어났다. 그리고 방대한 연산량을 가지고 있는 심층신경망의 문제를 해결하기 위해 고성능 컴퓨팅(HPC, High Performance Bomputing)과 GPU (Graphics Processing Unit)을 사용하여 딥러닝 기술이 발전하게 되었다[5].

GPU를 통한 딥러닝 분야의 발전과 함께 다양한 언어를 통해 딥러닝 프레임워크(Deep Learning Framework)가 구현되었다. 프레임워크는 응용 프로그램을 개발하기 위한 라이브러리나 모듈 등을 효율적으로 사용할 수 있도록 묶어 놓은 일종의 패키지로써 편리하게 효과적인 딥러닝 적용을 위해 다양한 딥러닝 프레임워크들이 만들어졌다. 초기에는 몬트리올, 버클리를 중심으로 한 대학교에서 연구용으로 개발되어 사용되었고 점차 확대되어 구글, 페이스북, 마이크로소프트사와 같은 글로벌 IT 기업이 가세하여 다양한 딥러닝 프레임워크가 개발되어 오픈소스 소프트웨어의 형태로 공유되고 있다.

딥러닝 기술은 고차원의 데이터에 대한 결과를 추론하는데 매우 효과적이어서 다양한 비즈니스 영역에 적용할 수 있는 것으로 알려져 있다. 예를 들면 이미지 인식, 음성 인식, 약물 분자 구성, 입자 가속기 데이터 분석, DNA 분석 등과 같은 영역에 대해서 다른 기계학습 알고리즘의 성능을 능가했다. 딥러닝은 자연어 이해, 주제 분류, 감정 분석과 같은 작업에도 의미있는 결과를 만들어냈다[6]. 딥러닝 프레임워크의 발전으로 다양한 인공지능(AI, Artificial Intelligence) 서비스들이 등장하고 있다. 특히 사물인터넷(IoT) 기반의 스마트시티는 방대한 양의 데이터에 대한 인공지능 서비스를 제공하고자 한다.

그러나 대부분의 인공지능 서비스들은 딥러닝 기술을 이용한 모델의 추론 정확도를 만족시킬 뿐 리소스를 효율적으로 사용하는데 한계점이 있다. 먼저, 인공지능을 기반으로 한 서비스를 실현하기 위해 GPU 기반 고속 병렬처리와 대규모 인프라 기술이 요구되어 인공지능 개발자와 인프라 엔지니어 간의 격차가 벌

어지고 인공지능 서비스 개발에 어려움이 생기고 있다[7]. 또한 실제 서비스 적용을 위해 웹 또는 플랫폼 아키텍처를 구성할 때에도 데이터 유입부터 전처리, 모델 학습, 서빙에 대한 요구를 다양한 인프라 기술을 운영하기 위한 비용이 발생된다.

그러므로 인공지능 서비스를 웹을 통해 제공하는 것에 대한 문제를 해결하기 위해 딥러닝 모델을 적용하는 안정적인 아키텍처에 대한 연구가 필요하다.

안정적인 아키텍처를 구성하기 위해 가상화 기술을 적용하고자 하는데 현재 가상화 기술 연구 분야에서는 하드웨어 레벨(Hardware-level) 가상화와 운영체제 레벨(OS-level) 가상화로 나뉘어 연구가 진행되고 있다. 대표적으로 도커(Docker)라는 서비스는 오픈소스 가상화 플랫폼으로 다양한 프로그램과 실행 및 개발 환경을 컨테이너(Container)로 추상화하고 동일한 인터페이스를 제공하여 프로그램의 개발과 배포 및 관리를 간단하게 만들어 준다. 도커는 운영체제 레벨 가상화 기술로써 단일 호스트 운영체제 위에 다중 애플리케이션 가상화 환경을 제공하는 컨테이너 방식을 이용한 기술을 제공하고 있다[8].

이에 따라 본 연구에서는 딥러닝 모델을 탑재한 안정적인 웹서비스 개발과 배포 및 관리를 위해 운영체제 레벨 가상화 기술을 제공하는 도커 컨테이너를 사용하고자 한다. 그리고 도커 컨테이너 기반의 딥러닝 모델 관리 시스템을 제공할 수 있는 다양한 아키텍처를 설계하고 아키텍처들 간의 성능을 비교하고자 한다.

2장에서는 딥러닝 프레임워크와 모델 서빙을 위한 기술들을 소개하고, 3장에서는 본 연구에서 제안하는 아키텍처들을 소개한다. 다음 4장에서는 제안한 아키텍처들에 대한 성능비교 실험에 대해 설명하고, 마지막으로 결론 및 향후과제를 제시한다.

Ⅱ. 이론적 배경

딥러닝 모델을 탑재한 인공지능 서비스를 구축하기 위해서는 다수의 사용자가 요청하는 작업에 대해 독립적인 프로세스에 할당되는 컴퓨팅 리소스의 관리가 필요하다. 이를 위해 다양한 모델을 적용할 수 있는 서빙 시스템과 관리 및 배포에 대한 비용을 줄일 수 있는 강력한 엔진이 필요하다[9]. 본 장에서는 웹서비스에서 딥러닝 모델 서빙을 운용할 때, 안정적인 아키텍처를 구성하기 위해 사용되는 기법들에 대해 개념을 중심으로 설명하고자 한다.

2.1 딥러닝 프레임워크 : 텐서플로우, 케라스

텐서플로우(TensorFlow)란 데이터 플로우 그래프(Data Flow Graph)를 사용하여 수치 연산을 하는 오픈소스 소프트웨어 라이브러리로써, 딥러닝 연구를 목적으로 구글에서 개발되었다. 텐서플로우는 기본 자료구조인 텐서(Tensor), 텐서의 수치 연산과정을 나타내는 엣지(Edge), 텐서를 처리하는 연산을 나타내는 노드(Node)로 구성되어 있다. 텐서플로우 아키텍처 구성의 유연성 덕분에 데스크탑, 서버, 모바일 디바이스에서도 CPU와 GPU를 사용하여 연산 작업 수행이 가능하다[10].

텐서플로우는 파이썬(Python) 라이브러리의 형태로 문법과 기능 측면에서 몬트리올 대학교에서 개발된 티아노(Theano)와 상당히 유사하다. 텐서플로우는 티아노와 비교하여 분산 컴퓨팅이 가능하다는 것과 추상화된 함수를 제공하여 딥러닝 모델 개발이 용이하다는 점에서 장점을 가지고 있다[11].

2.2 텐서플로우 서빙

텐서플로우 서빙(TFServing)은 딥러닝 모델을 서빙할 수 있도록 도와주는 시스템으로써 오픈소스로 공개되어 있어 사용자가 커스터마이징하여 클라우드(Cloud) 서버 혹은 이외의 환경에도 서빙할 수 있다. 텐서플로우 서빙 시스템이 지원하는 딥러닝 프레임워크의 종류는 텐서플로우로 학습된 모델에 한정되어 있지만, 모델 학습부터 모델 서빙, 모델 버전 관리 등 데이터 파이프라인 시스템과 통합을 지원하는 방식이 매우 유연하다. 또한 모델 조회 및 추론을 수행하는 시스템의 핵심 코드는 추가된 모델로 인해 발생하는 성능 저하를 피하기 위해 최적화시킨다. 성능 최적화를 통해 텐서플로우 서빙은 CPU 단일 코어 기준으로 초당 100,000건의 추론요청을 처리할 수 있으며, 처리 속도를 올리기 위해서 GPU를 사용할 수 있다[12,13].

<그림 1>은 텐서플로우 서빙의 내부 실행 시퀀스

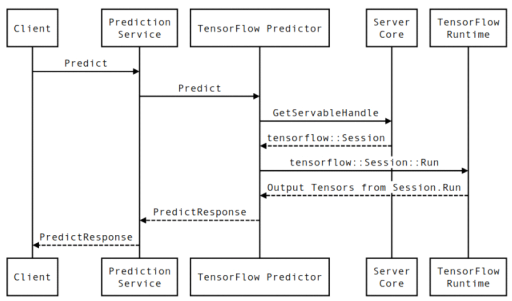


그림 1. 텐서플로우 서빙 내부 실행 시퀀스
Fig. 1. Internal Execution Sequence of TFServing

를 보여준다. 텐서플로우 서빙은 내부적으로 실시간에 최적화되어 있는 통신 방법을 사용함으로써 연산 처리 시간을 최소화한다. 서빙되는 텐서플로우 기반 모델은 C++로 빌드되어 제공되기 때문에 연산 처리 시간을 최소화한다. 입력데이터를 텐서플로우 서빙 모델로 전송하기 위해 REST(Representational State Transfer) 방식뿐만 아니라 gRPC(google Remote Procedure Call) 호출 인터페이스를 제공하며, 다양한 종류의 모델들을 동시에 수용할 수 있다.

프로토버프(Protocol Buffers)를 사용하여 데이터를 직렬화 하는 gRPC 방식은 JSON (JavaScript Object Notation) 방식보다 효율적으로 데이터를 전송한다. <그림 2>는 프로토버프와 JSON 간의 데이터 전송 속도를 비교한 실험결과이다. HTTP/2를 사용하는 gRPC의 페이로드(Payload)는 프로세스 간 통신 (IPC, Inter-process Communication) 시나리오에 기반한 아키텍처 구성 시 높은 성능을 기대 할 수 있다[14].

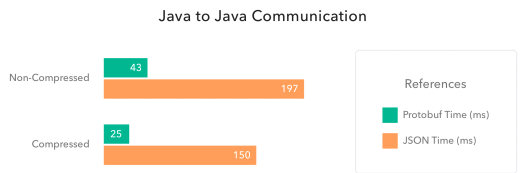


그림 2. 프로토버프와 JSON의 통신 속도 비교
Fig. 2. Comparison of Communication Speed Between Protobuff and JSON

2.3 도커 컨테이너

서론에서 간략하게 설명했듯이, 서버 분야에서 사용되는 가상화 기술은 크게 하드웨어 레벨 가상화와 운영체제 레벨 가상화가 있다. 하드웨어 레벨 가상화 기술은 <그림 3>에서 볼 수 있듯이, 가상화 계층에서 제공하는 가상환경은 실제 하드웨어에 적재된 운영체제 위에 위치하여 게스트 운영체제를 서비스한다. 이 기술은 가상화 계층 내부 핵심 구성 요소인 VMM(Virtual Machine Monitor)인데, 이 VMM은 물리적인 하드웨어를 활성화된 가상환경의 하드웨어에 연결하여 가상환경에서 호스트 운영체제가 사용하고 있는 하드웨어 자원을 사용 할 수 있도록 게스트 운영체제의 자원을 관리한다.

운영체제 레벨 가상화는 하나의 CPU에 2개 이상의 운영체제가 존재하는 하드웨어 레벨 가상화와는 다르게 하나의 운영체제만 수행되고, 단일 호스트 운영체제 위에 다중 애플리케이션이 실행될 수 있는

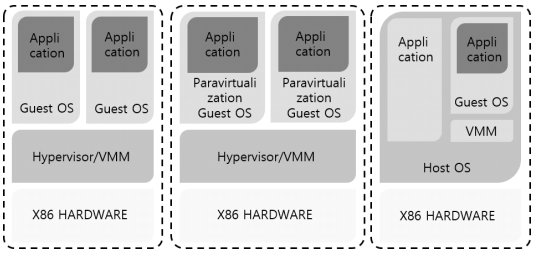


그림 3. 하드웨어 수준 가상화 종류
Fig. 3. Hardware-level Virtualization Types

컨테이너를 생성하여 어플리케이션 단위의 독립된 가상환경을 제공한다. 각각의 애플리케이션 컨테이너 안에는 호스트 운영체제와는 별개 운영체제와 네트워크, 프로세서 등을 생성한다. 운영체제 레벨 가상화 방법은 <그림 4>과 같이 컨테이너 방식과 하드웨어 에뮬레이터(Hardware Emulator) 방식으로 구분된다.

앞서 설명했던 하드웨어 레벨 가상화는 CPU 가상화 기술을 이용한 전가상화(Full-virtualizaion) 방식의 KVM(Kernel-based Virtual Machine)과 반가상화(Para-virtualizaion) 방식의 Xen으로 세부적으로 나뉘게 되는데 이러한 가상화 기법의 등장이 기존에 사용되었던 가상화 방식에 비해 성능을 향상 시킨 것은 사실이나, 전가상화와 반가상화는 호스트 운영체제 위에 추가적인 운영체제를 설치하기 때문에 결과적으로 성능 문제가 있었고 이를 개선하기 위해 프로세스를 격리하는 방식이 등장하였다.

이러한 방식을 리눅스에서는 컨테이너라고 하는데, 컨테이너 방식은 가상화 계층이 호스트 운영체제 위에 임베디드 형태로 구성되고, 각각의 애플리케이션별로 추가적인 컨테이너를 생성하는 방식이다. 각각의 컨테이너 안에 생성된 가상 운영체제는 호스트 운영

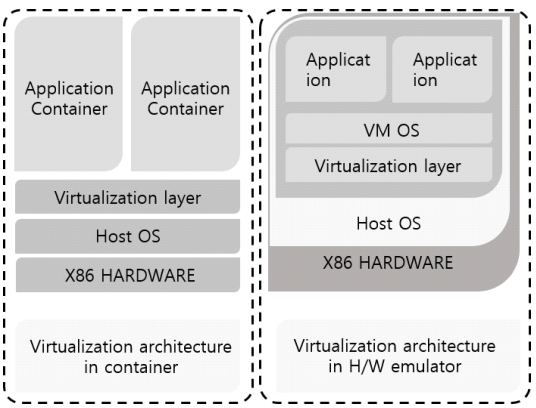


그림 4. 운영체제 수준 가상화 종류
Fig. 4. OS-level Virtualization Types

체제에 있는 커널을 공유하기 때문에 컨테이너는 그 자체가 호스트 운영체제의 자원을 일부 할당하여 작업을 수행하는 서버형태의 단일 프로세스로써 호스트 운영체제로부터 단순히 프로세스를 격리시키기 때문에 기존의 방식에 비해 가볍고 빠르게 동작할 수 있는 장점이 있다[15].

이러한 컨테이너 기법을 사용하여 개발된 상용 소프트웨어들 중 리눅스 컨테이너의 상징이라고 불리는 대표적인 제품인 도커(Docker)는 컨테이너 기반의 오픈소스 가상화 플랫폼으로써 다양한 프로그램과 실행환경을 포함하고 있는 이미지를 빌드하여 컨테이너로 추상화하고 동일한 인터페이스를 제공하여 개발자들의 프로그램의 배포 및 관리를 단순하게 해준다.

도커에서 가장 중요한 개념인 이미지는 컨테이너를 실행하기 위한 모든 정보를 포함하는 객체로써, 도커파일(Dockerfile)을 이용하여 이미지를 생성한다. 도커파일은 이미지를 생성할 때의 필요한 명령어의 집합으로 정의할 수 있으며, 설치해야하는 패키지, 소스코드, 컨테이너 구동과 동시에 실행 되어야하는 명령어와 셸 스크립트 등을 순차적으로 실행하여 사용자가 원하는 이미지를 만들 수 있도록 파일형태로 작성된다. 도커파일을 작성하여 빌드시킨 하나의 이미지를 기반으로 여러 컨테이너를 생성할 수 있고 컨테이너의 상태 변경 및 삭제 시에도 이미지는 원본 상태 그대로를 유지한다. 이러한 이미지를 기반 하여 생성된 컨테이너는 이미지를 실행한 상태라고 볼 수 있고 변경되는 사항은 컨테이너에 저장되며, 변경된 컨테이너 상태를 그대로 다시 이미지로 만들어서 업데이트 또한 가능하다. 컨테이너는 호스트 운영체제의 컴퓨팅 리소스를 일부 할당하여 실행이 되는데, CPU나 메모리는 컨테이너가 실행하는 프로세스가 필요한 만큼만 추가로 할당되며 리소스 재설정 및 자동 반영이 가능하기 때문에 성능 상의 손실도 거의 없다. 또한 하나의 서버에 여러 개의 컨테이너를 독립적으로 실행가능하며, 실행 중인 컨테이너에 접속하여 명령어를 입력할 수 있고, 추가적인 패키지 설치 및 호스트 운영체제의 특정 디렉토리를 컨테이너 내부 디렉토리로 연결하여 사용할 수도 있다.

이 외에도 도커는 호스트 운영체제 내에서 도커 네트워크 브릿지(Docker Network-bridge)를 사용하여 운영체제의 특정 포트로 포트포워딩을 지원함으로써 컨테이너간의 네트워크를 구성해 각각 독립적으로 실행되고 있는 컨테이너들을 유연하게 연결할 수 있다. 또한 이를 용이하게 구성하도록 도커-컴포즈(Docker-compose)를 제공하고, 추가로 컨테이너 오케스트레이션 도구인 도커-스웜(Docker-swarm) 등을 제공하여 유연하고 안정적인 관리 및 배포를 지원한다[16].

Ⅲ. 제안 아키텍처

3.1 아키텍처 종류

본 연구에서는 다양한 방법으로 구성된 4가지 아키텍처들을 제안한다. 또한 실험을 위해 데이터 파이프라인은 고려하지 않은 상태로 구성하였다.

3.1.1 Backend embedding

<그림 5>의 a 아키텍처는 웹 백엔드에서 직접 처리하는 방식의 아키텍처로써, 컨테이너 기반의 nginx 웹 서버를 사용하고 파이썬 기반 WAS(Web Application Server)인 Django 웹 프레임워크, Gunicorn 미들웨어를 포함하고 있다. 딥러닝 학습 모델을 Django 백엔드 부분에 함수로 삽입하여 실시간 추론을 하는 방식으로써 컨테이너 간 연결을 통해 실시간으로 데이터를 주고받는다.

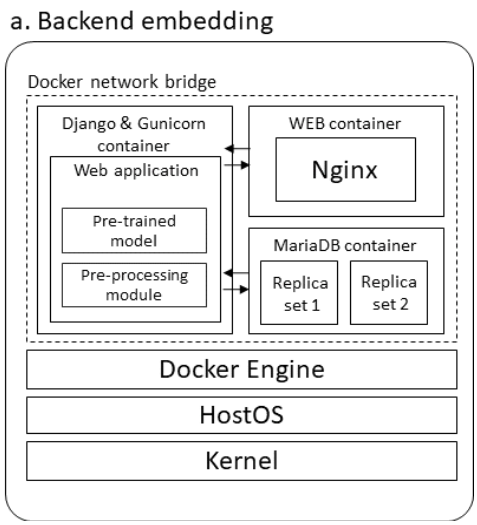


그림 5. 웹 백엔드 함수 내장형 모델 서빙 아키텍처
Fig. 5. Web-backend function embedded Model Serving Architecture

3.1.2 REST API

1) Flask

<그림 6>의 b 아키텍처는 플라스크 API를 이용해서 처리하는 방식의 아키텍처로써, a 아키텍처와 동일하게 도커 컨테이너 기반의 nginx, Django,

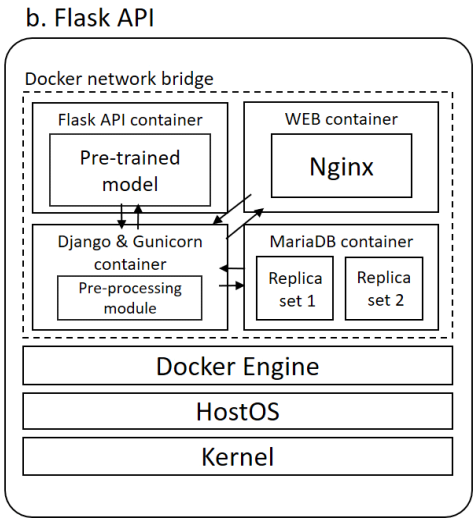


그림 6. 플라스크 API 기반 모델 서빙 아키텍처
Fig. 6. Flask API-based Model Serving Architecture

Gunicorn을 사용하지만, a 아키텍처와는 다르게 Flask라는 경량 웹서버 라이브러리를 사용하여 구축한 REST API 서버를 통해 딥러닝 학습 모델을 서빙 하는 아키텍처이다[17].

2) FastAPI

<그림 7>의 c 아키텍처는 b 아키텍처와 동일하게 파이썬 기반으로 개발한 REST API 방식의 모델 서빙 아키텍처이지만 node, Go 언어와 동등한 매우 높은 성능을 제공해주는 FastAPI 라이브러리를 활용하여

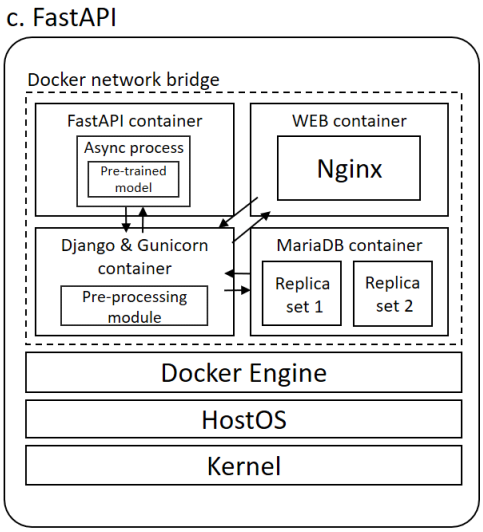


그림 7. 패스트 API 기반 모델 서빙 아키텍처
Fig. 7. FastAPI-based Model Serving Architecture

추론 프로세스를 실행할 수 있는 아키텍처이다[18].

3.1.3 TFserving

<그림 8>의 d 아키텍처는 도커 컨테이너 기반의 텐서플로우 서빙을 이용하는 방식의 아키텍처로써, 실시간 딥러닝 분석 결과를 출력하기 위해 딥러닝 학습 모델을 포함한 텐서플로우 서빙을 이용한다.

앞서 설명했듯이, 텐서플로우 서빙은 딥러닝 모델을 위한 확장성을 갖고 있는 고성능 모델 서빙 기술로써 사전 학습된 모델을 쉽게 배포하고 활용할 수 있고, 모델의 저장과 추론을 수행하기위해선 사용자가 로컬 또는 클라우드와 같은 물리적인 서버 위에서 사용하는 것이 요구된다[19].

위에서 설명한 4개 아키텍처의 컨테이너 간 데이터 전송은 도커 컨테이너 네트워크 브릿지를 사용하여 포워딩된 포트를 통해서 이루어지며, WEB 컨테이너를 통해서 접근하는 사용자 요청은 Gunicorn 미들웨어를 거쳐 WAS인 Django로 전달되어진다. 사용자가 요청하는 데이터는 a 아키텍처의 경우 WAS 내부의 호출된 모듈을 통해서 모듈 내부에 임베딩된 딥러닝 학습 모델에 전달되지만, b, c, d 아키텍처의 경우 데이터를 JSON 형식으로 변환 후 POST 방식의 통신을 통해 모델 서빙 서버에 전달되어 추론된 결과를 반환받는다.

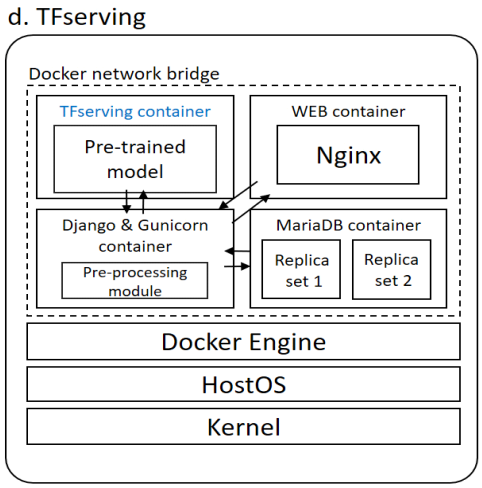


그림 8. 텐서플로우 서빙 기반 모델 서빙 아키텍처
Fig. 8. TFServing-based Model Serving Architecture

3.2 실험 데이터

본 연구에서 제안하는 아키텍처들 간의 성능 비교를 위해 CIFAR-10(Canadian Institute For Advanced

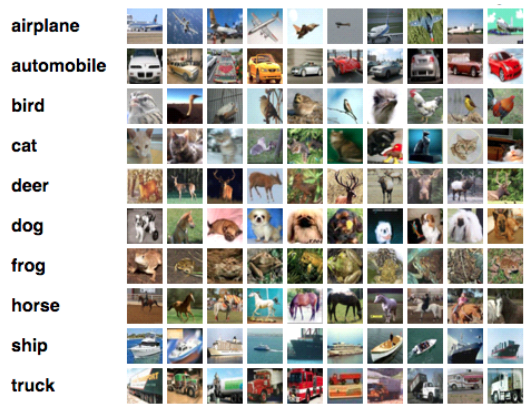


그림 9. 모델 학습 및 실험을 위한 CIFAR-10 데이터셋
Fig. 9. CIFAR-10 Dataset for Model Training and Experimentation

Research) 이미지 데이터를 사용하였다. CIFAR-10 데이터셋은 학습용 데이터 50,000건과 시험용 데이터 10,000건으로 구성되어 있으며, <그림 9>와 같이 총 10개의 클래스를 가지는 이미지 데이터를 가지고 있다. 클래스별 학습용 데이터와 시험용 데이터는 라벨별 학습용 데이터 5,000건과 시험용 데이터 1,000건으로 구성되어 있다. 해당 데이터는 Keras 라이브러리를 통해 다운로드할 수 있다.

3.3 실험 설계

제안 아키텍처들 간의 성능비교를 위한 CIFAR-10 데이터셋 분류 모델을 만들기 위해 이미지 인식에 좋은 성능을 보여주는 CNN(Convolutional Neural Network) 알고리즘을 이용하여 신경망을 구현하였다.

CNN은 딥러닝 알고리즘의 일종으로써 이미지 처리에 특화 되어있는 인공지능 신경망 종류인 합성곱 신경망이다. CNN은 특정 그림에서만 뉴런이 반응 하는 것을 모티브로 삼아, 입력된 이미지의 특정 영역(Local)의 특징을 추출하는 아이디어에서 착안되어 패턴이나 물체를 인식하는 생물의 시각처리과정을 모방한 모형으로써 LeCun에 의해 딥러닝 분야에서 활성화 되었다[20].

입력 계층과 출력 계층이 각각 한 개씩 존재하며, 입력 계층과 출력 계층 사이에 하나 이상의 합성곱 계층(Convolution Layer)과 풀링 계층(Pooling Layer) 그리고 완전 연결 계층(Fully-connected Layer)으로 구성되어 있다. CNN은 입력 계층을 통해 이미지를 입력하고 합성곱 계층을 통해 필터링 되는 과정을 거쳐 각 데이터가 갖고 있는 특징을 추출한다[21]. <그림 10>은 실험에서 사용될 CNN의 전체 구조를 나타낸다.

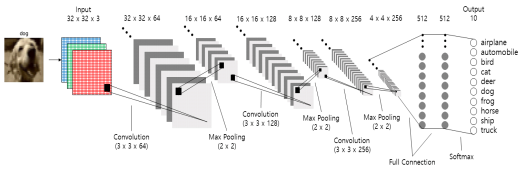


그림 10. 모델 학습 및 실험을 위한 CNN 구조
Fig. 10. CNN structure for Model Training and Experimentation

아래 <표 1>에서는 실험에서 사용할 CNN을 구성하는 계층(Layer)의 종류 및 크기와 활성화 함수 등을 확인할 수 있고, 실험을 위해 사용된 H/W, S/W 스펙은 다음 <표 2>에서 확인할 수 있다.

본 연구에서 실험하고자 하는 아키텍처들 간의 성능을 면밀하게 살펴보기 위해 앞서 설명한 CIFAR-10 학습용 데이터 50,000건에 대해 학습한 CNN모델을 아키텍처별로 동일하게 적용하여 실험을 진행했다. 아키텍처에 탑재된 CNN모델에 대한 시험용 데이터

표 1. CNN 구성 상세 정보
Table 1. Detail Explanations of CNN Configuration

Layer	Size	Filter		Activation Function
Input	32×32×3	Convoltion	3×3×64	relu
		Dropout	0.5	
Layer-1	32×32×64	Convoltion	3×3×64	relu
		Dropout	0.5	
Layer-2	32×32×64	Max Pooling	2×2	-
Layer-3	16×16×64	Convoltion	3×3×128	relu
		Dropout	0.5	
Layer-4	16×16×128	Max Pooling	2×2	-
Layer-5	8×8×128	Convoltion	3×3×256	relu
		Dropout	0.5	
Layer-6	8×8×256	Max Pooling	2×2	-
Layer-7	4×4×256	Flatten	-	relu
Layer-8	512	Dense	512	relu
Layer-9	512	Dense	10	softmax
Output	10	-		-

표 2. 하드웨어 제원 및 소프트웨어 버전 정보
Table 2. H/W Specifications and S/W Version Information

H/W	CPU		RAM	GPU	
	Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz		64.0GB	GeForce RTX 2070 SUPER	
S/W	OS	Python	Tensor Flow	CUDA	cuDNN
	Windows 10 Pro	3.8.3	2.3.0	10.1	7.6.5.32

10,000건의 요청 방식은 Embedded Function 아키텍처에서는 일반적인 함수를 실행하는 방법으로 진행하고 Flask API, Fast API, TFServing 아키텍처들은 REST 방식으로 데이터를 처리하였다. 데이터 10,000건 처리에 대한 실험을 아키텍처별 각각 10번씩 실행하여 나온 평균 처리 시간, 시스템 리소스 사용량 등을 파악하여 실험 결과를 도출하였다.

3.4 실험 결과

앞에서 설명한 실험 설계를 통해 아키텍처별 시험용 데이터 10,000건 처리 실험을 수행하였다. <표 3>에는 실험한 아키텍처들의 데이터 개수별 처리 시간 값에 대한 표이다.

실험 결과에 따르면 데이터 처리 속도는 TFServing, Embedded function, Fast API, Flask API 순으로 나타나는 것을 알 수 있다. 특히 TFServing 아키텍처는 10,000건 데이터 처리에 대해 평균 82.519초로 다른 아키텍처들에 비해 월등한 처리속도를 보여주었다. 데이터 개수별로 확인했을 경우에도 가장 빠르게 데이터를 처리하고 있음을 알 수 있었다.

REST API 방식인 Fast API와 Flask API의 성능을 비교하면 Fast API가 약 36~39배 정도의 성능 차이를 보였고, Fast API와 Embedded Function을 비교했을 때는 Embedded Function이 약 1.5~1.9배 정도의 성능 차이를 보였다.

데이터 개수별 처리 시간에 대한 성능 비교 실험결과를 확인했을 때는 위의 설명과 같이 TFServing 아키텍처를 사용하는 것이 트래픽에 대한 처리를 안정적으로 처리할 수 있을 것으로 판단된다.

다음으로 각 아키텍처 간 종합적인 비교를 위해 데이터 10,000건 처리 시 할당되는 CPU, RAM과 같은 자원 및 아키텍처들의 종합적인 정보를 <표 4>과 같이 정리하였다.

표에서 확인 가능하듯이 각 아키텍처가 서빙 할 수 있는 모델의 형태는 TFserving과 다른 세 아키텍처가

표 3. 데이터 크기에 따른 평균 요청 처리 속도 비교
Table 3. Comparison of Average Request Processing Speed by Data Size

Number of data	Embedded function	Flask API	Fast API	TFServing
1	0.038 s	2.090 s	0.056 s	0.009 s
10	0.296 s	20.851 s	0.577 s	0.082 s
100	2.927 s	209.449 s	5.763 s	0.826 s
1,000	37.840 s	2059.050 s	56.694 s	8.167 s
10,000	289.219 s	20702.672 s	531.959 s	82.519 s

표 4. 아키텍처별 종합 정보
Table 4. Comprehensive Information Of Each Architecture

Architecture	Type of model	language	Type of request	System resource	
				CPU (%)	RAM (MB)
Embedded function	general	python	None	21.7	450.5
Flask API	general	python	REST	5.4	200.7
Fast API	general	python	REST	18.2	260.2
TFServing	tensorflow servable	C++	REST, gRPC	23.7	275.6

상이한 것을 확인할 수 있는데, 파이썬으로 개발된 아키텍처는 강력하고 유연한 형태로 분석 라이브러리를 제공하는 개발언어의 특성을 이용하여 다양한 딥러닝 프레임워크로 학습된 딥러닝 모델을 서빙할 수 있는 반면, TFServing 아키텍처는 C++로 빌드되어 내부 코드의 커스터마이징을 지원하지 않고 텐서플로우 프레임워크를 사용하여 학습시킨 모델(Tensorflow Servable)만을 서빙할 수 있다는 한계점이 있다.

하지만, TFServing 아키텍처가 보유하고 있는 성능은 특정 모델 타입만을 지원한다는 제한사항에도 불구하고 다른 아키텍처들과 비교 하였을 때 압도적인 성능 차이를 보여주고 있다. TFServing의 경우 REST API 호출 형태로 개발된 아키텍처들과 동일하게 REST 호출 인터페이스를 제공하지만, 이외에도 앞서 문헌연구에서 언급했듯이 gRPC 인터페이스를 제공하기 때문에, gRPC 형식의 호출을 통해 같은 실험을 진행 한다면 현재 도출된 결과보다 더 향상된 성능을 나타낼 것으로 보여진다.

<그림 11>은 아키텍처별 데이터 10,000건을 처리하는 동안 측정된 평균 CPU 사용률과 평균 RAM 사용량을 나타낸다. 결과를 살펴보면 8코어 CPU 기준으로 Flask API가 5.4%로 사용률이 가장 낮고, TFServing이 23.7%로 사용률이 가장 높은 것으로 확

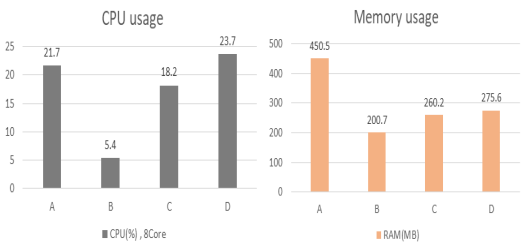


그림 11. 아키텍처별 평균 시스템 자원 사용량 비교
Fig. 11. Comparison of Average System Resource Usage by Each Architectures

인되었다. CPU 사용률은 TFServing, Embedded Function, Fast API, Flask API 순으로 높고, 이것은 데이터 처리 속도와 비례하는 양상을 확인할 수 있다.

또한 RAM 사용량의 경우는 Embedded Function, TFServing, Fast API, Flask API 순으로 Flask API가 200.7MB로 가장 낮고, Embedded Function이 450.5MB로 가장 높은 것으로 확인되었다.

두 결과를 종합해보면 Embedded Function의 경우 실제 CPU 사용률은 두 번째로 높고 RAM 사용량은 두 번째인 TFserving 대비 1.63배 높게 나온다. 만약 Embedded Function 아키텍처를 이용하여 인공지능 서비스를 구성할 경우 WAS(web application server)에 대한 시스템 리소스를 많이 할당하는 것이 유리할 것으로 예상된다. 딥러닝 모델에 대한 사용량이 늘어날 것을 가정한다면 도커 컨테이너 기반으로 리소스를 분배하기 때문에 TFServing, Fast API, Flask API의 경우를 이용한 방식이 안정적인 서비스를 제공하는데 유리할 것으로 예상된다.

Ⅳ. 결 론

본 연구에서는 스마트시티와 같이 딥러닝 모델을 기반으로 하는 인공지능 서비스를 안정적으로 제공하기 위해 다양한 웹서비스 아키텍처를 설계하고 아키텍처들 간의 성능을 비교하였다. 웹서비스를 개발할 때 만들 수 있는 아키텍처 중 웹백엔드에 모델을 로드하는 함수를 직접 개발하는 Embedded function, REST API를 만들어 모델 결과 값을 받는 방식을 이용한 Python 경량화 웹프레임워크 Flask API와 Fast API, TensorFlow에서 제공하는 서빙 방식인 TFServing까지 총 4가지 아키텍처를 예시로 만들고 실험을 진행하였다. 실험 결과 데이터 처리 속도 면에서는 TFServing이 가장 뛰어난 성능을 보여주었지만 탑재 가능한 모델의 종류, 커스터마이징 면을 고려했을 때에는 충분한 속도를 보여주는 Fast API도 안정적인 아키텍처로 개발하기에 충분하다는 결과를 보여주었다.

본 연구가 가지는 학술적 의의는 인공지능 서비스를 구축하는 아키텍처들에 대한 비교분석을 시도했다라는 점에서 의의를 갖는다. 실무적으로는 인공지능 서비스를 개발하려는 개발자들에게 다양한 방법론을 제시하였다는 점에서 의의를 갖는다.

그러나 다음과 같이 몇 가지 한계점을 갖는다. 먼저 딥러닝 알고리즘 종류, 구조에 따라 연산량의 차이가 있는데 본 연구에서는 CNN 모델 한 가지만을 적용함으로써 다양한 인공지능 서비스 구축 예시를 보여주지 못했다는 한계점을 갖고 있다. 그러므로 향후 연구에서는 연산량의 차이에 대한 분석과 함께 다양한 모델을 적용하여 상황과 여건에 맞는 최적의 아키텍처를 구축할 수 있는 연구가 필요하다.

두 번째는 딥러닝 모델 결과값 도출을 위해 데이터 전처리에 대한 부분은 통제하여 진행하였기 때문에 데이터 전처리에 따른 시스템 리소스 사용 차이, 처리 속도 등을 고려하지 못했다는 점에서 한계점을 갖는다. 따라서 향후 연구에서는 데이터를 전처리에 대한 연산량과 처리속도 등을 파악하는 연구가 필요하다.

세 번째로 본 연구는 데이터 파이프라인과 다양한 머신러닝 배포를 쉽게 만들어주는 플랫폼들과는 비교하지 못했다는 점에서 한계점을 갖는다. Tensor Flow 뿐만 아니라 Scikit-learn, XGBoost와 같은 머신러닝 라이브러리들도 배포 가능한 Kubeflow와 대용량 분산처리와 머신러닝을 같이 적용하는 Spark와 같은 플랫폼들과 비교하는 연구가 필요하다.

마지막으로 실제 사용되고 있는 웹서비스 또는 플랫폼에 직접 적용한 것이 아니라 특정 실험 조건을 만들어 실험을 진행하였기 때문에 제안하는 아키텍처들에 대한 일반성이 충분이 검증되지 못하였다. 따라서 실제 서비스를 배포할 때 고려되는 로드 밸런싱과 같은 방법도 적용하여 제안 아키텍처들의 일반성을 검증하는 것이 향후 연구과제가 될 것이다.

References

[1] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE Trans. Pattern Anal. and Mach. Intell.*, vol. 35, no. 8, pp. 1798-1828, Mar. 2013.

[2] W. K. Jung, J. U. Kim, T. T. Dao, J. H. Park, J. Y. Park, J. H. Shin, J. H. Jung, G. W. Jo, H. H. Kim, H. W. Nam, and J. J. Lee, "Hardware and software support for deep learning," *KIISE*, vol. 34, no. 9, pp. 10-20, Sep. 2016.

[3] H. Y. Choi and Y. H. Min, "Introduction to deep learning and major issues," *KIPS Rev.*, vol. 22, no. 1, pp. 7-21, Jan. 2015.

[4] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786,

pp. 504-507, Jul. 2006.
[5] Y. S. Lee and P. J. Moon, “A comparison and analysis of deep learning framework,” *J. KIECS*, vol. 12, no. 1, pp. 115-122, Feb. 2017.
[6] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436-444, May 2015.
[7] K. S. Jang and J. H. Kim, “A study on high performance gpu based container cloud system supporting TensorFlow serving deployment service,” in *Proc. KIPS Conf.*, pp. 386-388, Jeju Island, Korea, Nov. 2017.
[8] https://docs.docker.com
[9] Y. G. Yim and H. W. Jin, “Analysis of impact of multi-core CPU bandwidth in docker containers,” *KIISE Trans. Computing Practices*, vol. 24, no. 12, pp. 675-680, Dec. 2018.
[10] https://www.tensorflow.org
[11] Y. Chung, S. M. Ahn, J. Yang, and J. J. Lee, “Comparison of deep learning frameworks : About theano, tensorflow, and cognitive toolkit,” *J. Intell. and Inf. Syst.*, vol. 23, no. 2, pp. 1-17, Jun. 2017.
[12] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar, “TensorFlow-Serving: Flexible, high-performance ML serving,” *Wkshps. ML Syst. at NIPS*, California, USA, Dec. 2017.
[13] J. T. Park, H. G. Kim, and I. Y. Moon, “Design and implementation of web compiler for learning of artificial intelligence,” *J. Advanced Navig. Technol.*, vol. 21, no. 6, pp. 674-679, Dec. 2017.
[14] https://auth0.com/blog/beating-json-performance-with-protobuf/
[15] Y. M. Bae, S. J. Jung, and W. Y. Soh, “Comparative analysis of the virtual machine and containers methods through the web server configuration,” *J. KIICE*, vol. 18, no. 11, pp. 2670-2677, Nov. 2014.
[16] R. Y. Jang, R. Lee, M. W. Park, and S. H. Lee, “Development of an AI analysis service system based on OpenFaaS,” *J. KCA*, vol. 20, no. 7, pp. 97-106, Jul. 2020.
[17] A. Yaganteeswarudu, “Multi disease prediction model by using machine learning and flask API,” *ICCES*, pp. 1242-1246, Coimbatore, India, Jun. 2020.
[18] https://fastapi.tiangolo.com/
[19] R. Chard, Z. Li, K. Chard, L. Ward, Y. Babuji, A. Woodard, S. Tuecke, B. Blaiszik, M. J. Franklin, and I. Foster, “DLHub: Model and data serving for science,” *Int. Symp. Parall. and Distrib. Process.* (IPDPS), pp. 283-292, Rio de Janeiro, Brazil, May 2019.
[20] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, “Gradient-based learning applied to document recognition,” in *Proc. IEEE*, vol. 86, no. 11, pp. 2278-2324, Nov. 1998.
[21] M. S. Lee and H. C. Ahn, “A time series graph based convolutional neural network model for effective input variable pattern learning : Application to the prediction of stock market,” *J. Intell. Inf. Syst.*, vol. 24, no. 1, pp. 167-181, Mar. 2018.

이 모 세 (Mo-se Lee)

2014년 2월 : 국민대학교 기계시스템공학 졸업
2019년 2월 : 국민대학교 비즈니스IT전문대학원 공학석사
2020년 : 이노커스 연구원
2021년~현재 : 브이알에듀 책임연구원

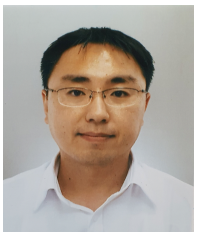
<관심분야> Data Engineering, Machine Learning, Deep Learning, AI Machine Vision
[ORCID:0000-0003-4919-1867]

강 민 수 (Min-su Kang)

2020년 2월 : 한서대학교 항공소프트웨어공학과 졸업
2020년 2월 : 이노커스 연구원
2021년~현재 : 브이알에듀 선임연구원

<관심분야> Deep learning, Data engineering, Linux, System Architecture
[ORCID:0000-0002-7855-1342]

김 인 호 (In-ho Kim)



2000년 2월 : 호서대학교 컴퓨터공학과 학사 졸업
2002년 2월 : 호서대학교 벤처대학원 중퇴
<관심분야> IoT, SMART HOME, 멀티미디어
[ORCID:0000-0002-6571-1634]

김 재 헌 (Jae-hun Kim)

2009년 3월 : 와세다대학교 경제학과 졸업
2017년 2월 : 고려사이버대학교 실용외국어학과 졸업
2019년 2월 : 고려대학교 교려대학원 중일지역비교문화 석사 수료
<관심분야> 마케팅, 클라우드, 인공지능, 빅데이터
[ORCID:0000-0002-2218-0976]