학사학위 청구논문

딥러닝을 통한 이미지 기반 식단 분석

(Image-based Diet Analysis Using Deep-learning)

2021년 12월 18일

숭실대학교 IT대학

전자정보공학부 IT융합전공

고 은 혜

학사학위 청구논문

딥러닝을 통한 이미지 기반 식단 분석

(Image-based Diet Analysis Using Deep-learning)

지도교수 : 신 요 안

이 논문을 학사학위 논문으로 제출함

2021년 12월 18일

숭실대학교 IT대학

전자정보공학부 IT융합전공

고 은 혜

(인준서)

고은혜의 학사학위 논문을 인준함

심사위원장 김종훈 (인)

심 사 위 원 신요안 (인)

2021년 12월 18일

숭실대학교 IT대학

목 차

표 및 그림 목차

국문초록

딥러닝을 통한 이미지 기반 식단 분석

전자정보공학부 IT융합전공 고은혜
지도교수 신요안

분류(Classification)에 주로 사용되는 머신러닝 알고리즘에는 KNN(K-Nearnest Neighbor), 의사결정 트리(Decision Tree), Random Forest, SVM(Support Vector Machine) 등이 대표적이며 머신러닝에서 더욱 심화되어 컴퓨터가 여러 처리 계층을 이용해 패턴을 인식하여 스스로 학습하도록 훈련시키는 기술인 딥러닝 중에서는 중요한 픽셀만을 사용하여 학습시키는 CNN(Convolutional Nerual Network)이 대표적인 분류 알고리즘이다. CNN 이전에는 이미지를 1차원 배열로 변환한 뒤 Fully Connected Neural Network로 학습시켰다. 하지만 이러한 방법은 이미지를 변환하는 과정에서 정보의 손실이 일어날 수밖에 없기 때문에, 이러한 단점을 보완하여 정보를 유지한 채로 학습을 진행하는 CNN이 등장하게 되었다. 이 중에서도 이미지 분류에 사용되는 대표적인 모델로는 AlexNet, ResNet, GoogleNet 등이 있다. 그 중에서도 EfficientNet은 모델의 너비(Width), 깊이(Depth), 입력 이미지의 해상도(Resolution)를 모두 고려하는 Compound scaling 방법을 제안한다.

본 논문에서는 Pytorch를 이용하여 EfficientNet-B0 모델을 만든 후 batch size와 learning rate, epochs를 변화시키면서 높은 정확도를 구현하였고, 그 이후에는 구현한 모델을 활용하여 식단 일기 작성 어플리케이션을 제작하였다.

I. 서 론

최근 들어 AI나 자율주행 등 딥러닝에 대한 관심이 굉장히 뜨거워지면서 이와 관련하여 여러 방면으로 활발하게 연구가 이루어지고 있다. 딥러닝에 대해 간단히 설명하면 ANN(Artificial Neural Network)에서 레이어의 숫자를 확장시킨 형태인데, 인간의 두뇌에서 뉴런들이 작동하는 방식으로 작동한다고 하여 인공신경망이라고 부른다. 딥러닝은 이러한 머신러닝 기법들 중에 하나로, 대표적으로 CNN과 RNN 알고리즘들이 주로 사용되고 있다. 딥러닝 알고리즘은 데이터로부터 머신을 학습시키는데, 헬스케어나, e-commerce, 광부 등의 분야에서 다양하게 사용되고 있다.

특히 최근에는 코로나 이후로 다이어트 및 건강에 대한 관심이 증가하여 식단 관리 및 관련 어플리케이션의 사용자가 증가하고 있으며, 본인 또한 이를 사용하며 겪었던 번거로움을 해소하고자, 딥러닝 모델을 통해 자동으로 메뉴를 인식하고 메뉴의 이름, 칼로리 등을 자동으로 입력해주는 어플리케이션을 개발하게 되었다.

서론에 이어 2장에서는 CNN과 핵심 모델이라고 할 수 있는 EfficientNet의 이론에 대하여 기술하고, 3장에서는 EfficientNet을 기반으로 모델을 설계하고 결과를 관찰한 후, 4장에서는 구현한 모델을 이용한 이미지 인식 식단 어플리케이션 제작을, 마지막으로 5장에서는 결론을 맺도록 한다.

Ⅱ. 딥러닝 이론

Ⅱ-1. CNN

CNN이란 Convolutional Neural Network의 약자로, 한국어로는 합성곱 신경망이라고 불린다. 딥러닝에서 주로 이미지나 영상 데이터를 처리할 때 사용하며, 전처리 작업이 들어가는 신경망 모델로 합성곱층(Convolution layer)과 풀링층(Pooling layer)을 Fully connected layer 이전에 추가하며 위 과정을 반복하여 이미지의 위치나 각도 변화에 대응하여 인식하게 된다. 대표적인 모델로는 AlexNet, ResNet, GoogleNet이 있다.

Ⅱ-2. EfficientNet

Neural Network의 성능을 높이기 위한 가장 쉬운 방법은 모델의 너비(Width), 깊이(Depth), 입력 이미지의 해상도(Resolution)를 키우는 것이었다. 하지만 각각의 요소를 하나씩 키웠을 때의 성능 향상은 그림 n에서 보는 것과 같이 다소 저조한 양상을 보임을 알 수 있다.

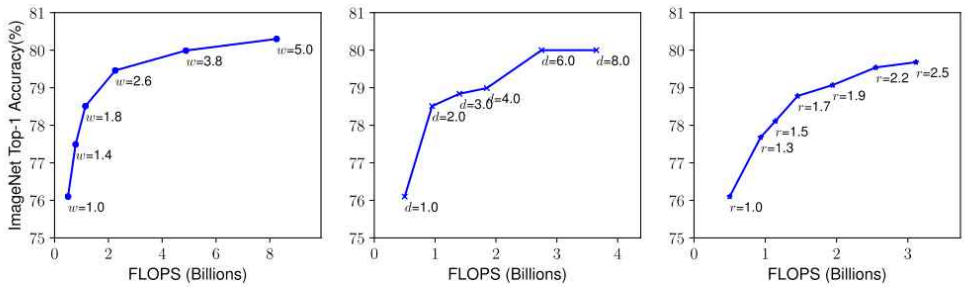

Figure 3. **Scaling Up a Baseline Model with Different Network Width (w), Depth (d), and Resolution (r) Coefficients.** Bigger networks with larger width, depth, or resolution tend to achieve higher accuracy, but the accuracy gain quickly saturate after reaching 80%, demonstrating the limitation of single dimension scaling. Baseline network is described in Table 1.

그림 2.1 : Single Dimension Model Scaling 실험 결과

지금까지 위의 3가지 요소를 동시에 고려하는 경우는 없었고, 'EfficientNet, Rethinking Model Scaling for Convolutional Neural Networks' 에서 처음으로 3가지 요소를 동시에 고려하는 Compound scaling을 제안한다. Compound scaling은 고정된 상수 값의 비율로 각각의 요소를 증가시키면 보다 좋은 성능을 보인다.

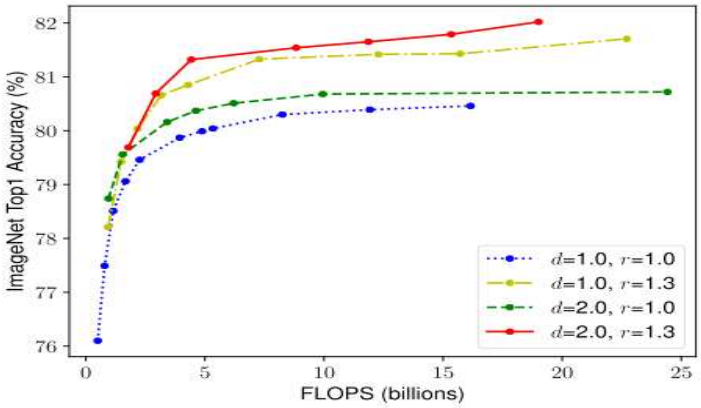

Figure 4. **Scaling Network Width for Different Baseline Networks**. Each dot in a line denotes a model with different width coefficient (w). All baseline networks are from Table 1. The first baseline network (d=1.0, r=1.0) has 18 convolutional layers with resolution 224x224, while the last baseline (d=2.0, r=1.3) has 36 layers with resolution 299x299.

그림 2.2 : Scaling Network Width 결과

$$\text{depth: } d = \alpha^\phi$$
$$\text{width: } w = \beta^\phi$$
$$\text{resolution: } r = \gamma^\phi$$
$$\text{s.t. } \alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$$
$$\alpha \geq 1, \beta \geq 1, \gamma \geq 1$$

그림 2.3 : Compound scaling에서 사용되는 변수 표기법

Depth, Width, Resolution은 각각 $\alpha \cdot \beta^2 \cdot \gamma^2 \approx 2$를 만족시켜야한다. 이때, Width와 Resolution에 제곱이 붙은 이유는 Depth는 키울 때 FLOPS도 같이 선형적으로 증가하지만, Width와 Resolution은 FLOPS가 제곱 배로 증가하기 때문이다. 이렇게 제한된 범위 내에서 찾은 변수의 비율 중에서, 가장 작은 모델이 바로 EfficientNet-B0이며 EfficientNet의 구조는 MNas Net과 유사하다. 다음 그림 n은 ImageNet을 활용하여 EfficientNet을 수행한 결과이다.

Table 1. **EfficientNet-B0 baseline network** – Each row describes a stage i with $\hat{L}_i$ layers, with input resolution $\langle \hat{H}_i, \hat{W}_i \rangle$ and output channels $\hat{C}_i$. Notations are adopted from equation 2.

Stage i	Operator $\hat{\mathcal{F}}_i$	Resolution $\hat{H}_i \times \hat{W}_i$	#Channels $\hat{C}_i$	#Layers $\hat{L}_i$
1	Conv3x3	224×224	32	1
2	MBConv1, k3x3	112×112	16	1
3	MBConv6, k3x3	112×112	24	2
4	MBConv6, k5x5	56×56	40	2
5	MBConv6, k3x3	28×28	80	3
6	MBConv6, k5x5	28×28	112	3
7	MBConv6, k5x5	14×14	192	4
8	MBConv6, k3x3	7×7	320	1
9	Conv1x1 & Pooling & FC	7×7	1280	1

그림 2.4 : EfficientNet-B0 구조

Table 2. **EfficientNet Performance Results on ImageNet** (Russakovsky et al., 2015). All EfficientNet models are scaled from our baseline EfficientNet-B0 using different compound coefficient ϕ in Equation 3. ConvNets with similar top-1/top-5 accuracy are grouped together for efficiency comparison. Our scaled EfficientNet models consistently reduce parameters and FLOPS by an order of magnitude (up to 8.4x parameter reduction and up to 16x FLOPS reduction) than existing ConvNets.

Model	Top-1 Acc.	Top-5 Acc.	#Params	Ratio-to-EfficientNet	#FLOPS	Ratio-to-EfficientNet
EfficientNet-B0	**76.3%**	**93.2%**	**5.3M**	**1x**	**0.39B**	**1x**
ResNet-50 (He et al., 2016)	76.0%	93.0%	26M	4.9x	4.1B	11x
DenseNet-169 (Huang et al., 2017)	76.2%	93.2%	14M	2.6x	3.5B	8.9x
EfficientNet-B1	**78.8%**	**94.4%**	**7.8M**	**1x**	**0.70B**	**1x**
ResNet-152 (He et al., 2016)	77.8%	93.8%	60M	7.6x	11B	16x
DenseNet-264 (Huang et al., 2017)	77.9%	93.9%	34M	4.3x	6.0B	8.6x
Inception-v3 (Szegedy et al., 2016)	78.8%	94.4%	24M	3.0x	5.7B	8.1x
Xception (Chollet, 2017)	79.0%	94.5%	23M	3.0x	8.4B	12x
EfficientNet-B2	**79.8%**	**94.9%**	**9.2M**	**1x**	**1.0B**	**1x**
Inception-v4 (Szegedy et al., 2017)	80.0%	95.0%	48M	5.2x	13B	13x
Inception-resnet-v2 (Szegedy et al., 2017)	80.1%	95.1%	56M	6.1x	13B	13x
EfficientNet-B3	**81.1%**	**95.5%**	**12M**	**1x**	**1.8B**	**1x**
ResNeXt-101 (Xie et al., 2017)	80.9%	95.6%	84M	7.0x	32B	18x
PolyNet (Zhang et al., 2017)	81.3%	95.8%	92M	7.7x	35B	19x
EfficientNet-B4	**82.6%**	**96.3%**	**19M**	**1x**	**4.2B**	**1x**
SENet (Hu et al., 2018)	82.7%	96.2%	146M	7.7x	42B	10x
NASNet-A (Zoph et al., 2018)	82.7%	96.2%	89M	4.7x	24B	5.7x
AmoebaNet-A (Real et al., 2019)	82.8%	96.1%	87M	4.6x	23B	5.5x
PNASNet (Liu et al., 2018)	82.9%	96.2%	86M	4.5x	23B	6.0x
EfficientNet-B5	**83.3%**	**96.7%**	**30M**	**1x**	**9.9B**	**1x**
AmoebaNet-C (Cubuk et al., 2019)	83.5%	96.5%	155M	5.2x	41B	4.1x
EfficientNet-B6	**84.0%**	**96.9%**	**43M**	**1x**	**19B**	**1x**
EfficientNet-B7	**84.4%**	**97.1%**	**66M**	**1x**	**37B**	**1x**
GPipe (Huang et al., 2018)	84.3%	97.0%	557M	8.4x	-	-

We omit ensemble and multi-crop models (Hu et al., 2018), or models pretrained on 3.5B Instagram images (Mahajan et al., 2018).

그림 2.5 : ImageNet을 활용한 EfficientNet 수행 결과

그림 n을 보면, 기존 Convolution Networks들에 비해 비슷한 정확도를 보이면서도 parameter수와 FLOPS가 굉장히 많이 줄어든 것을 확인할 수 있다. 또한 기존에 ImageNet에서 가장 높은 정확도를 보였던 GPipe보다도 높은 정확도를 보이는 것을 확인할 수 있다.

III. EfficientNet 기반 모델 구현 및 결과

앞선 표에서 본 바와 같이 EfficientNet에도 여러 개의 모델이 존재하지만, 본 논문의 목적은 식단 분석의 어플리케이션을 개발하는 것이기 때문에 가장 가벼운 모델인 EfficientNet-B0를 선택하여 구현을 진행하였다.

Batch size	128		150		180	
Learning rate	0.1	0,05	0.1	0.05	0.1	0.05
Loss	0.7	0.66	0.68	0.66	0.66	0.6
Accuracy	83.5	84.57	83.7	84.95	84	85.23

표 3.1 : Batch size와 Learning rate에 따른 Loss 및 Accuracy 비교

구현에 사용된 GPU는 NVIDIA의 Quardo RTX 6000이고 Epochs는 200회로 동일하게 진행하였다. 결과적으로 Batch size가 128이고 Learning rate가 0.1일 때 가장 낮은 정확도를 보였고, Batch size가 180이고 Learning rate가 0.05일 때 가장 높은 정확도를 보였다.

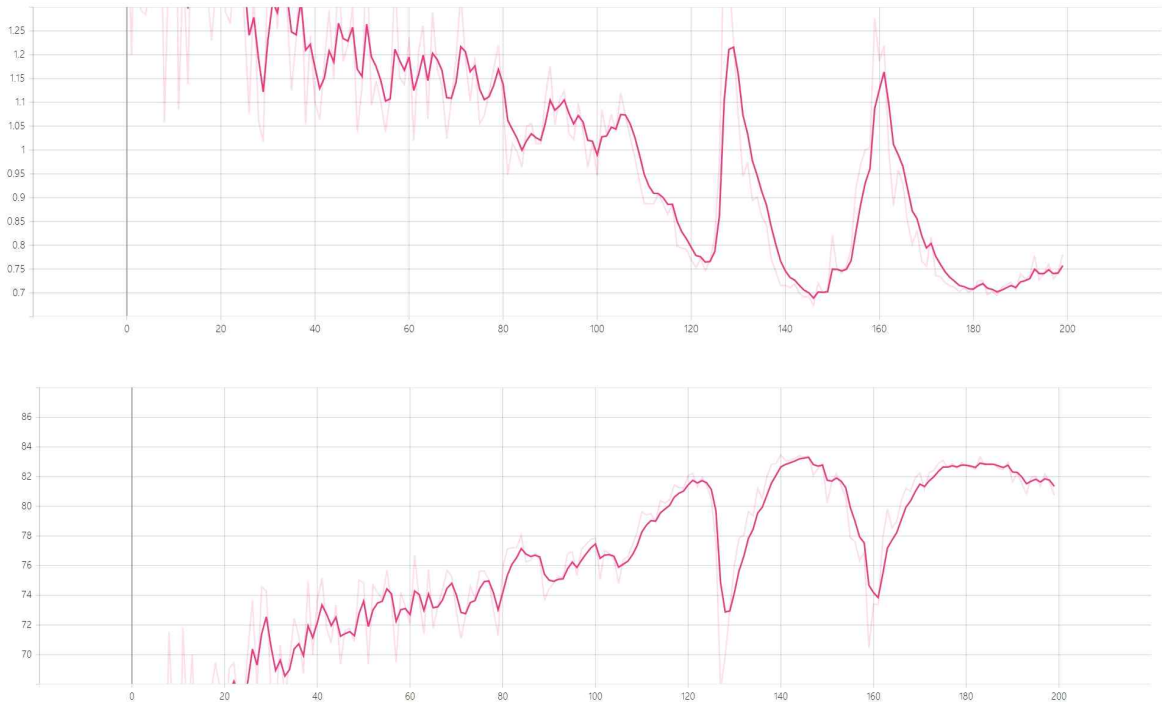

그림 3.1 : batch 128, lr 0.1의 loss 및 accuracy 그래프

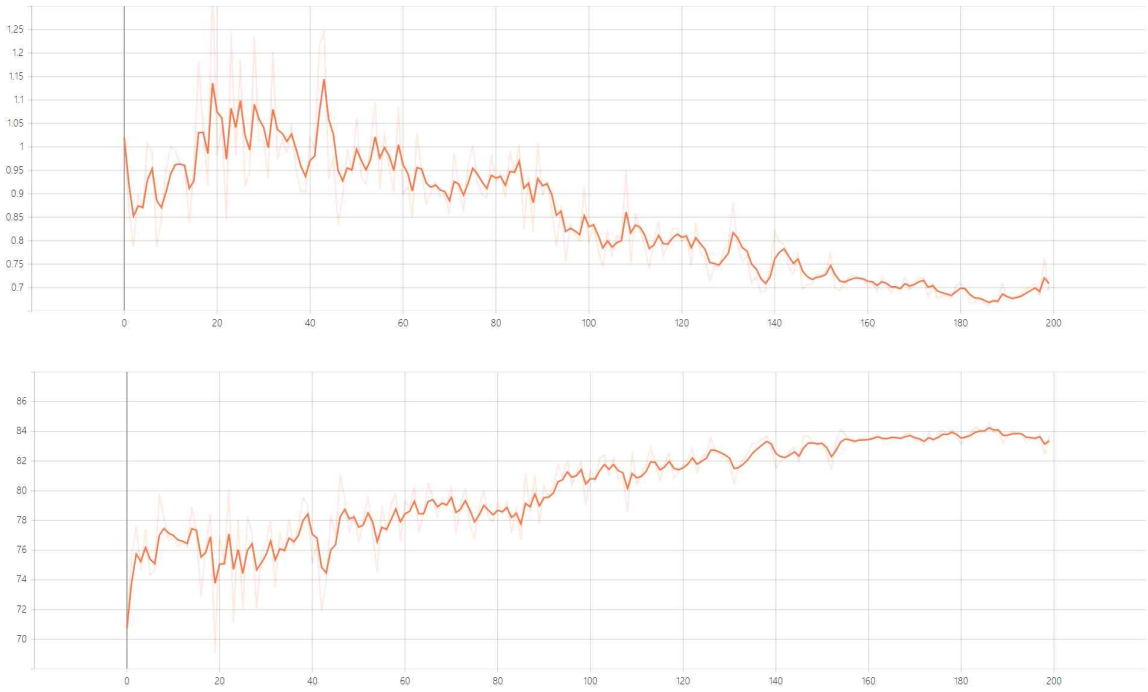

그림 3.2 : batch 128, lr 0.05의 loss 및 accuracy 그래프

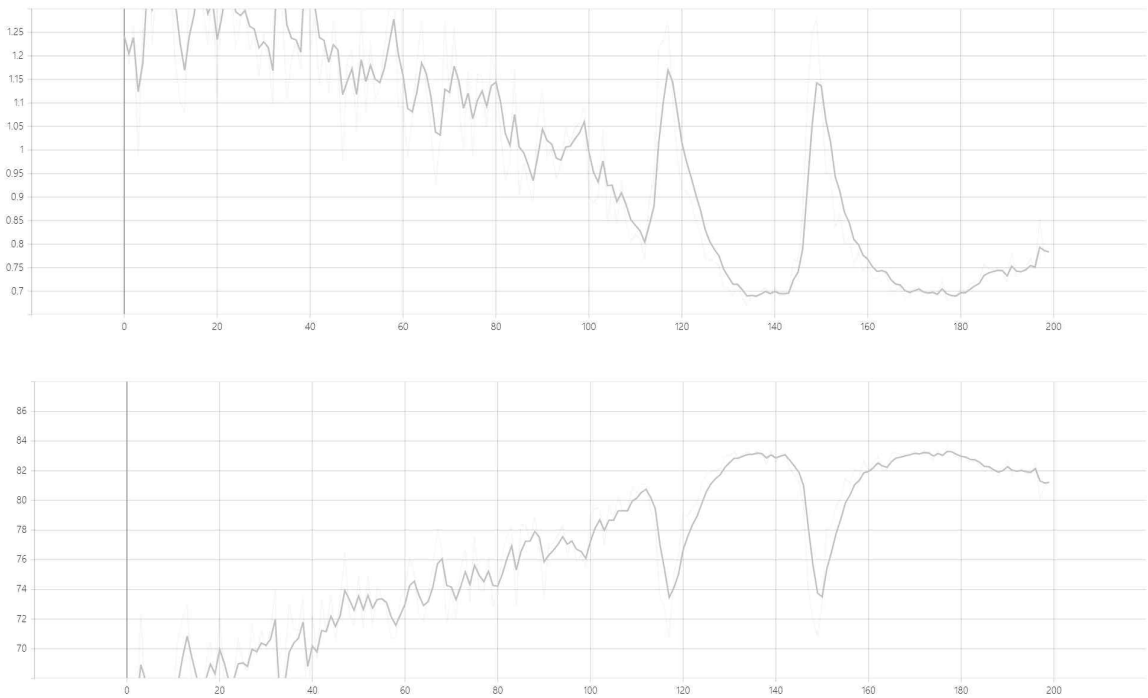

그림 3.3 : batch 150, lr 0.1의 loss 및 accuracy 그래프

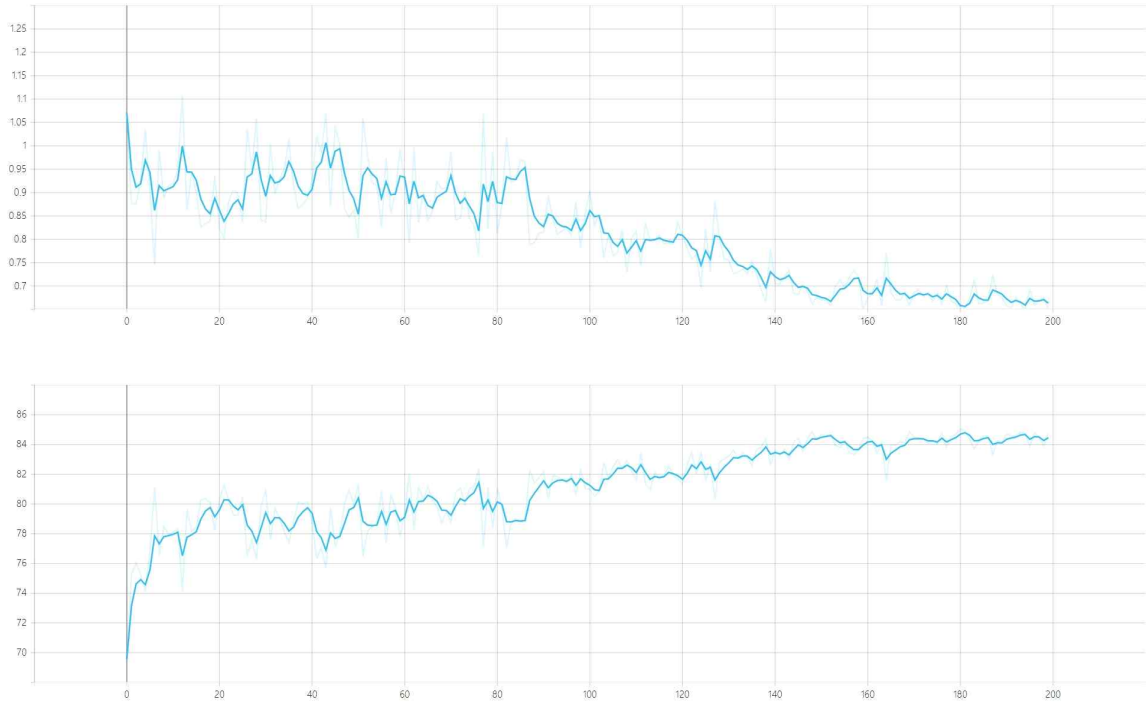

그림 3.4 : batch 150, lr 0.05의 loss 및 accuracy 그래프

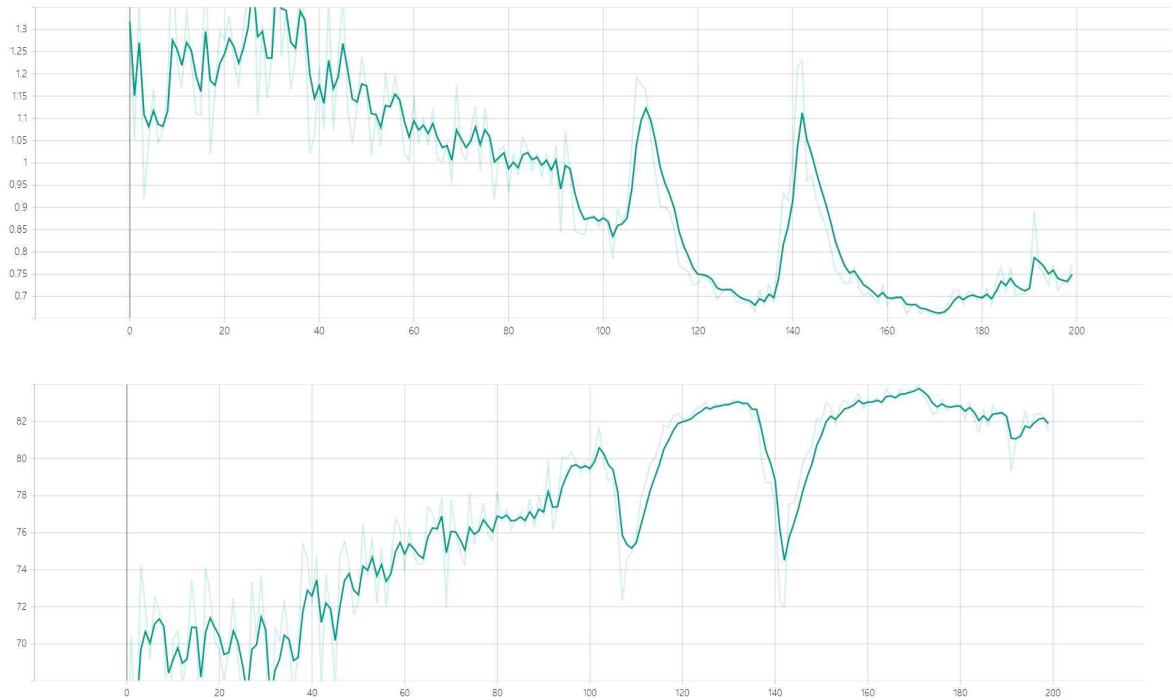

그림 3.5 : batch 180, lr 0.1의 loss 및 accuracy 그래프

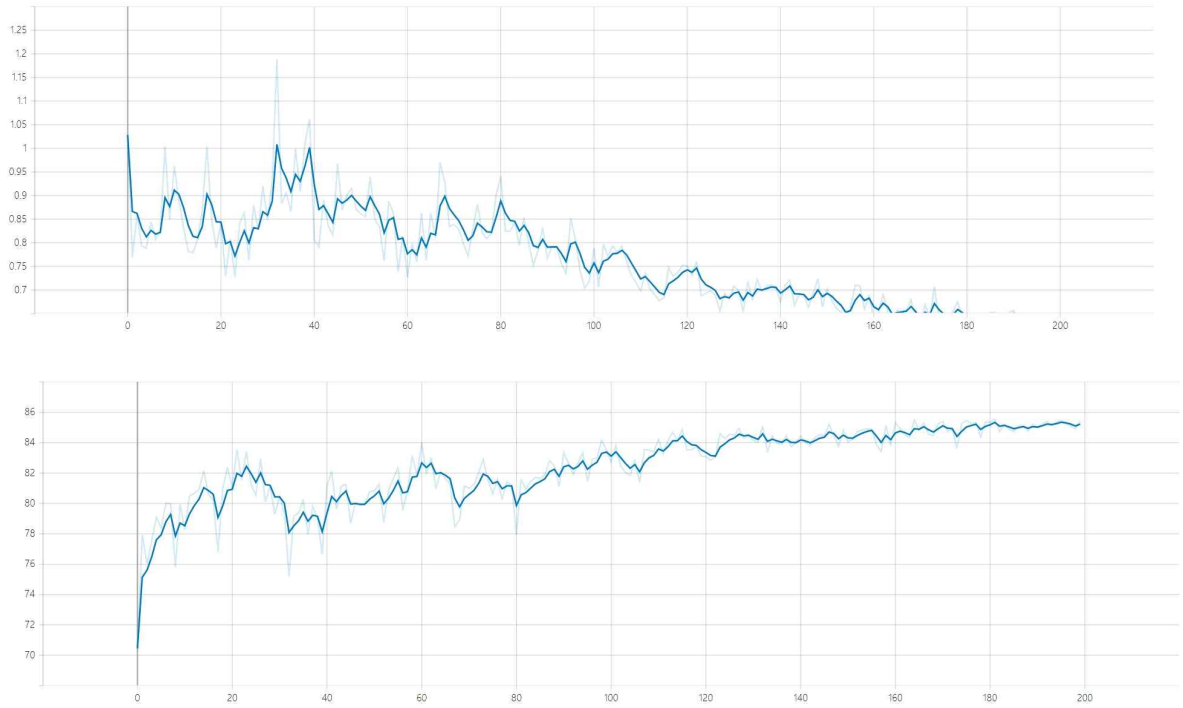

그림 3.6 : batch 180, lr 0.05의 loss 및 accuracy 그래프

그 이후, 보다 나은 모델을 구현하고자 최적 모델에 100회를 추가로 학습을 진행하였으나 의미 있는 결과를 보이지는 못 하였다. 총 300회의 학습 후, 최종적인 최저 Loss는 0.62이고 최고 Accuracy는 85.82이다.

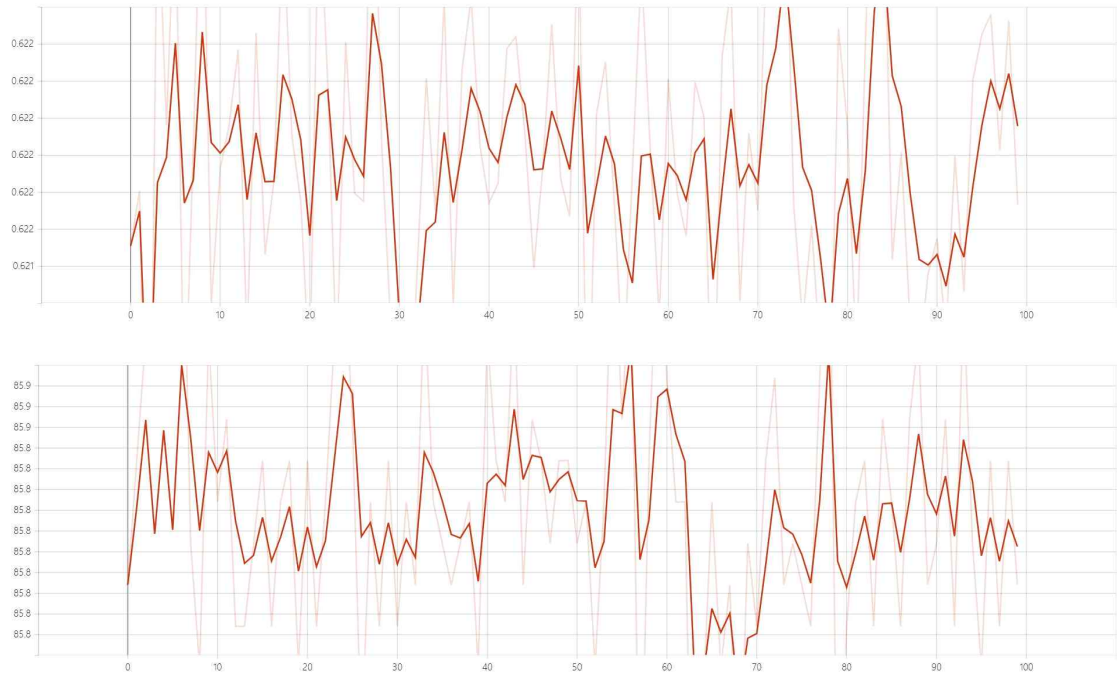

그림 3.7 : batch 180, lr 0.05, epochs 300의 loss 및 accuracy 그래프

어플리케이션에 모델을 업로드 하기 전, 입력 값으로 음식 사진이 들어왔을 때 제대로 예측하는지 확인하는 과정을 진행하였다. 본 논문에서 사용한 EfficientNet_Pytorch 라이브러리에는 예측을 진행하는 predict 함수가 존재하지 않았기 때문에, 직접 구현하여 진행하였다.

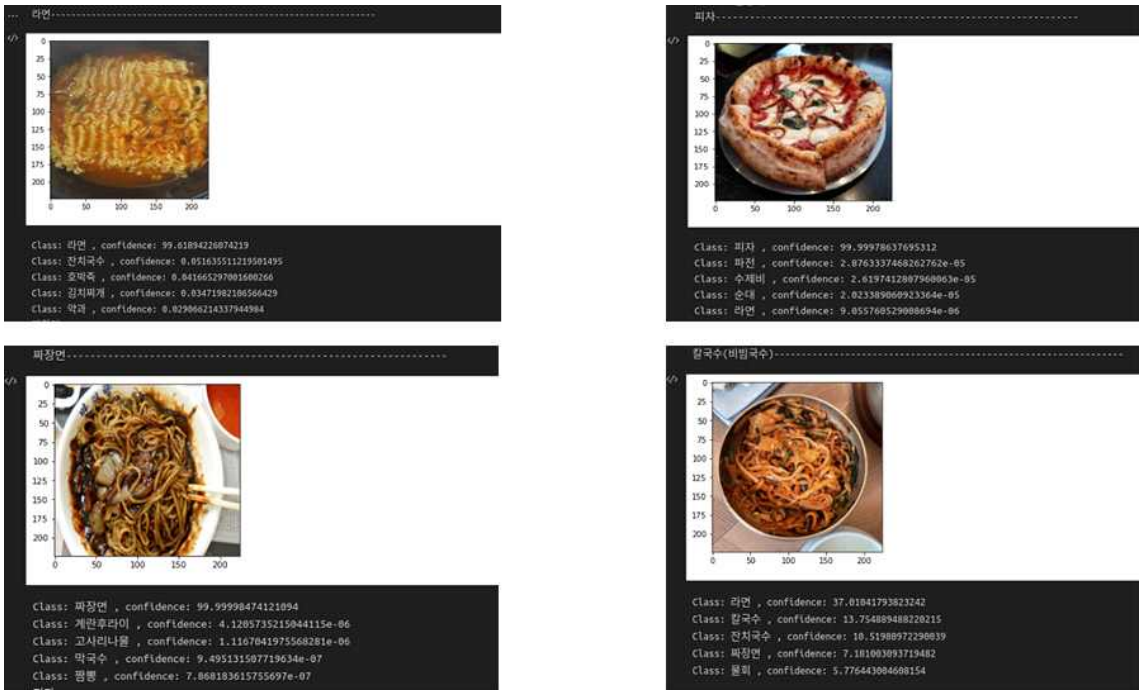

그림 3.8 : EfficientNet을 이용한 메뉴 예측 결과

Ⅳ. 어플리케이션 제작 및 구현

Ⅳ-1. 제작 환경

앞에서 구현한 모델을 이용하여 이미지 인식 식단 다이어리 어플리케이션 제작을 진행하였다. 본 논문에서는 ios 운영체제 어플리케이션으로 제작하였기에 XCode와 SwiftUI 프레임워크를 제작 환경으로 선택하여 개발을 진행하였다.

SwiftUI는 드래그 앤 드롭 방식의 형태로, 미리 보기창에 추가하면 자동으로 코딩이 되는 방식이므로 코딩을 쉽게 할 수 있고, 코딩한 내용을 바로 확인할 수 있는 장점이 있다.

또한, 리액트 네이티브(React Native - facebook에서 개발된 오픈소스 모바일 애플리케이션 프레임워크) 나 플러터(Flutter - 구글에서 개발된 오픈소스 모바일 애플리케이션 프레임워크) 와 같이 MVVM(Model View View Model) 패턴으로 화면을 구성하였다. MVVM 패턴에서는 view를 통해 액션이 들어오면, view model 로 액션을 전달하고, 이후 view model 에서 model 에게 데이터를 요청하는데, model 에서 응답받은 데이터를 view model 에 가공하여 저장한다. 이후 view model 에서 데이터 바인딩을 하여 view 에 데이터를 전송하는 구조이다. 이 패턴을 사용하면 모듈 간 의존성을 없앨 수 있는데 따라서 데이터 로딩이나 화면 구성이 빠른 장점이 있다.

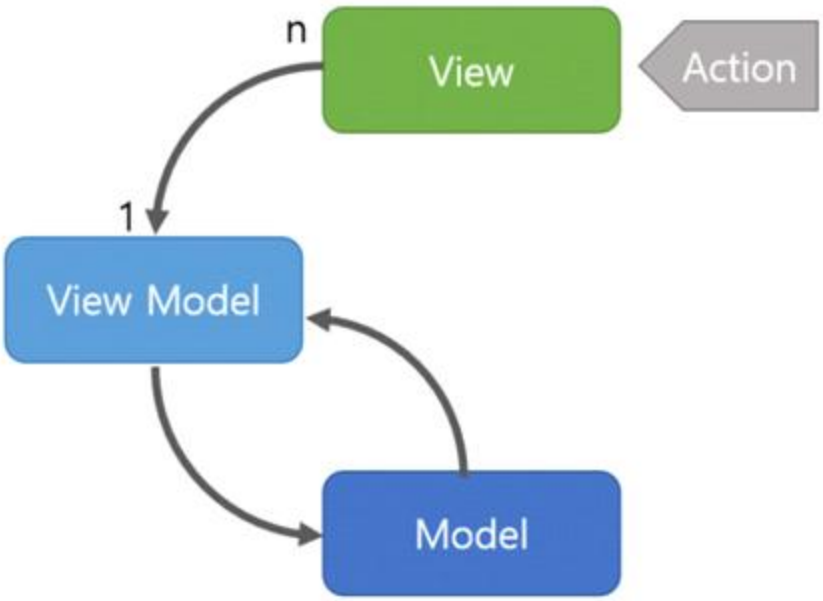

그림 4.1 : MVVM 패턴을 도식화한 그림

Ⅳ-2. Pytorch to TFlite

본 논문에서 학습시킨 efficient net 모델을 핸드폰에서도 실행할 수 있도록 하기 위해 몇 가지의 과정들을 진행하였다. 모델을 어플리케이션에 적용하기 위해 tensorflowLite(기기 내 추론을 위한 오픈소스 딥러닝 플랫폼) 를 사용하여 tf파일로 변환하였는데 앞에서 efficient net모델을 pytorch로 구현하였기 때문에 tensorflowLite를 사용하기 위해서 tensorflow로 모델을 변환하였다.

우선 훈련된 pytorch 모델을 개방형 표준형식인 onnx로 내보냈다.

```
#pytorch->onnx

with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['test']):
            inputs= inputs.to(device)
            #labels = labels.to(device)
model.eval()
model.set_swish(memory_efficient=False)
print('Model image size: ', model._global_params.image_size)

torch.onnx.export(model, inputs, "model_180_0.05_300.onnx", verbose=True)
```

그림 4.2 : pytorch to onnx

이후, ONNX 를 tensorflow로 변환하였다.

```
#onnx->tf
import onnx

model = onnx.load("model_180_0.05_300.onnx")

onnx.checker.check_model(model)

onnx.helper.printable_graph(model.graph)
```

```
#onnx->tf
from onnx_tf.backend import prepare

TF_PATH = "./model_180_0.05_300.pb"
ONNX_PATH = "./model_180_0.05_300.onnx"
onnx_model = onnx.load(ONNX_PATH)

tf_rep = prepare(onnx_model)
tf_rep.export_graph(TF_PATH)
```

그림 4.3 : onnx to tf

마지막으로 tf파일을 tensorflowLite 를 이용하여 앱 적용이 가능하게 하였다.

```python
#tf->tflite
import tensorflow as tf

TF_PATH = "./model_180_0.05_300.pb"
TFLITE_PATH = "./model_180_0.05_300.tflite"
os.environ["PATH"] = "/home/cip-server/PythonHome/myenv"

converter = tf.lite.TFLiteConverter.from_saved_model(TF_PATH)

converter.experimental_new_converter = True

converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS,
                                       tf.lite.OpsSet.SELECT_TF_OPS]

tf_lite_model = converter.convert()

open(TFLITE_PATH, "wb").write(tf_lite_model)

```

그림 4.4 : tf to tflite

Ⅳ-3. 어플리케이션 구현

구현하고자 하는 식단다이어리 어플리케이션의 기본적인 기능은 아래와 같다.

1. 사진을 인식하여 음식의 종류를 인식
2. 인식한 음식의 칼로리 정보 제공
3. 이 정보들을 달력에 기록

칼로리 정보 제공에는 총 칼로리, 탄수화물, 단백질, 지방에 대한 정보가 있다. 이 데이터들을 json파일로 만들어 앞에서 구현한 모델과 연결할 수 있도록 하였다.

그림 4.5 : 칼로리 json 파일

앱 디자인 툴로는 ios 앱 개발을 할 때, 연동이 유용한 Adobe XD 툴을 사용하여 어플리케이션 디자인을 진행하였다.

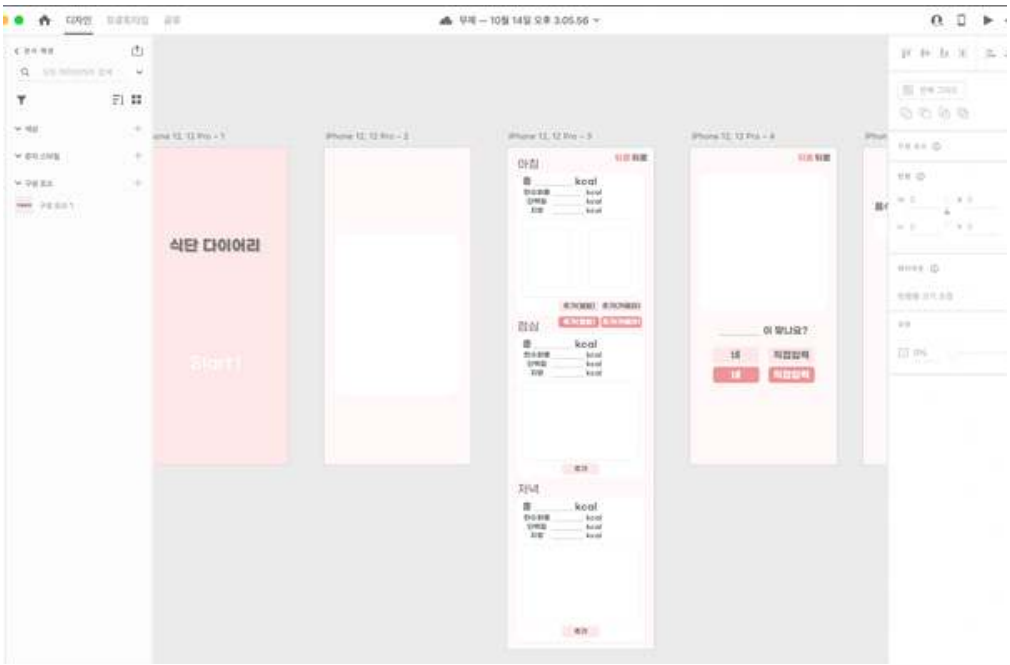

그림 4.6 : 전체적인 식단 다이어리 어플리케이션 디자인

입력 사진으로는 카메라로 찍은 사진과 앨범에서 선택하는 사진 모두 이용할 수 있도록 하였다. 사진을 입력 받으면 앞에서 학습시킨 딥러닝 모델을 통해 음식 종류를 식별한다. 음식 종류를 식별한 후에는 json파일을 통해 탄수화물, 단백질, 지방, 총 칼로리 정보를 받아와 다이어리에 사진과 함께 입력할 수 있도록 하였다. 이때, 인식을 잘 하지 못할 경우를 대비하여 사용자가 직접 음식을 입력하는 기능을 사용하여 칼로리를 입력할 수도 있도록 하였다.

그림 4.7 : 기능별 화면 디자인

그림 4.8 : 사용자가 직접 기록하는 기능 화면

딥러닝을 동작시키는 방식으로는 tflite 파일을 구글 파이어베이스(Firebase) api를 이용하여 적용하도록 하였다. 이 방식은 로컬에서 모델을 로딩하는 것이 아니라 구글 클라우드에서 모델을 호스팅하는 방식으로 이를 통해 딥러닝 계산을 빠르게 처리하도록 하였다.

아래는 어플리케이션을 구현한 후 실제 구현된 모습이다.

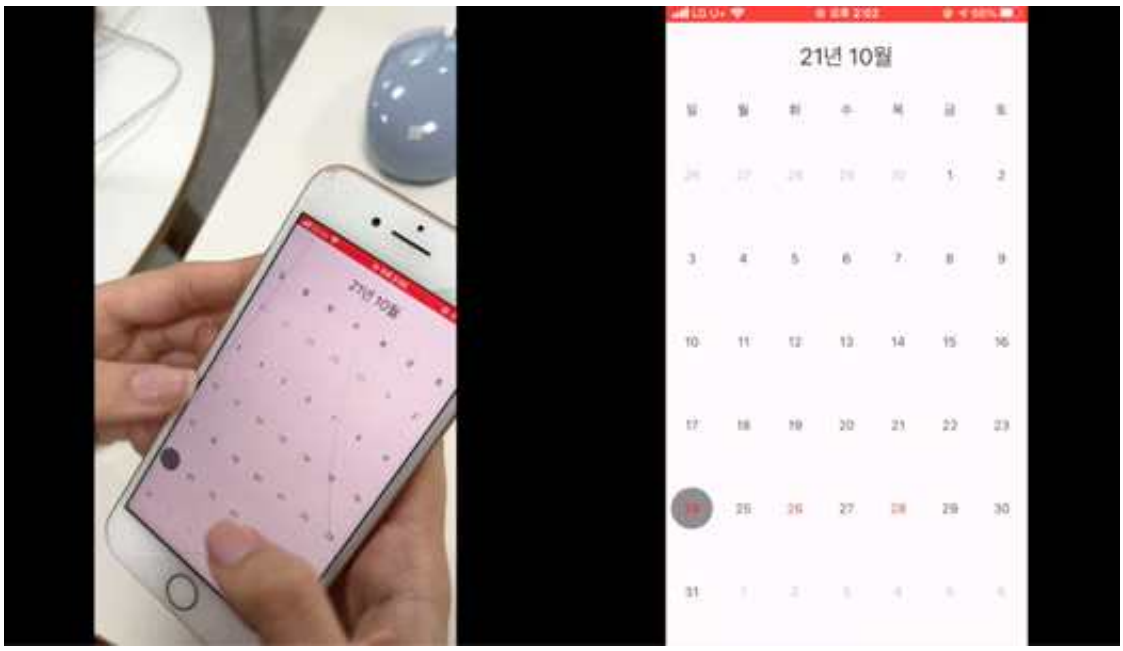

그림 4.9 : 구현된 달력 모습

기능1 : 앨범 속 사진 인식

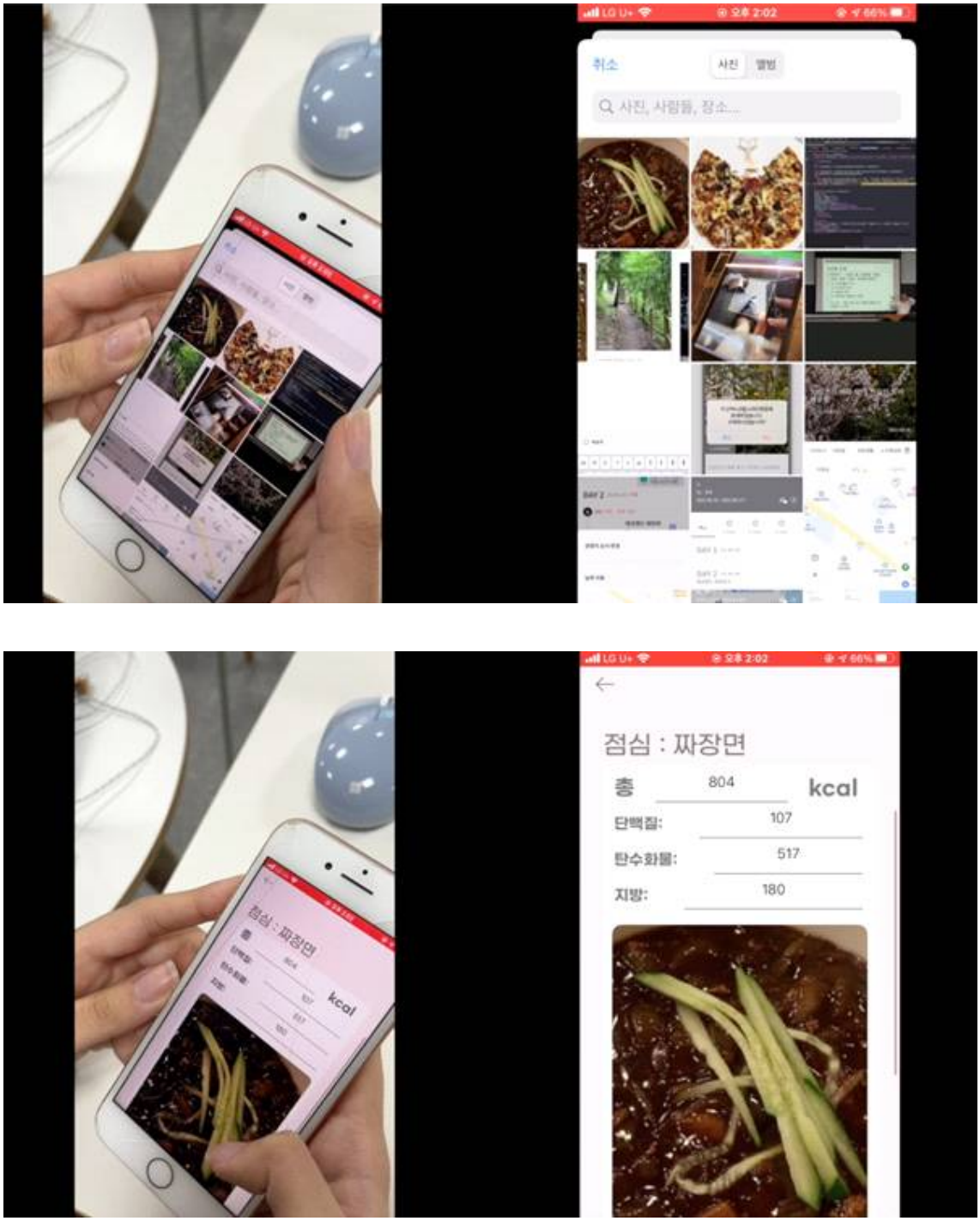

그림 4.10 : 기능 1 사진

휴대폰 앨범에 저장되어 있던 사진(짜장면)을 불러온 후 인식하여 칼로리를 저장하는 모습이다.

기능 2 : 카메라를 통한 사진 입력

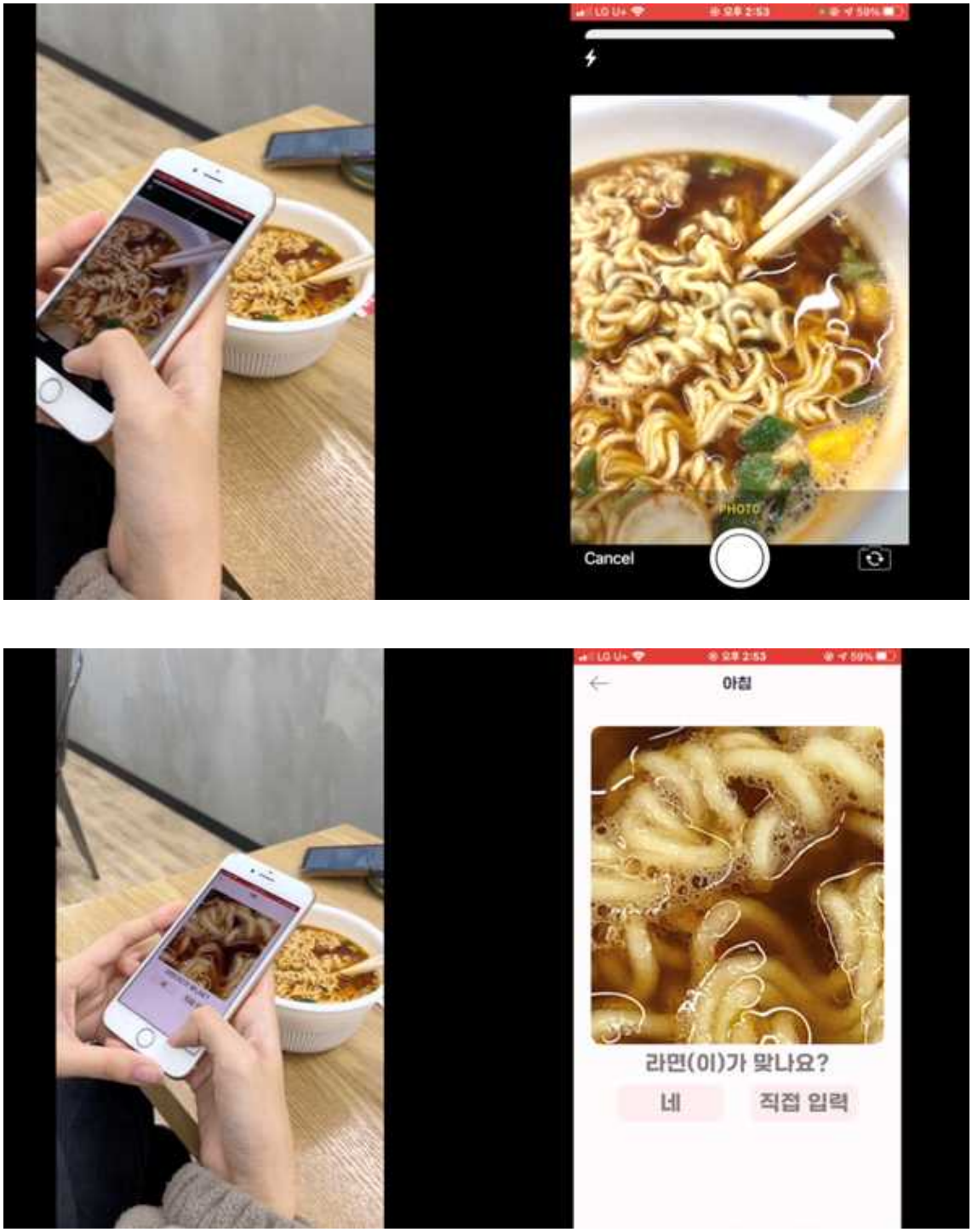

그림 4.11 : 기능 2 사진

휴대폰 카메라와 연동되어 사진을 찍고, 그 이미지를 인식하여 음식 종류를 인식하는 모습이다.

기능 3 : 직접 입력 기능

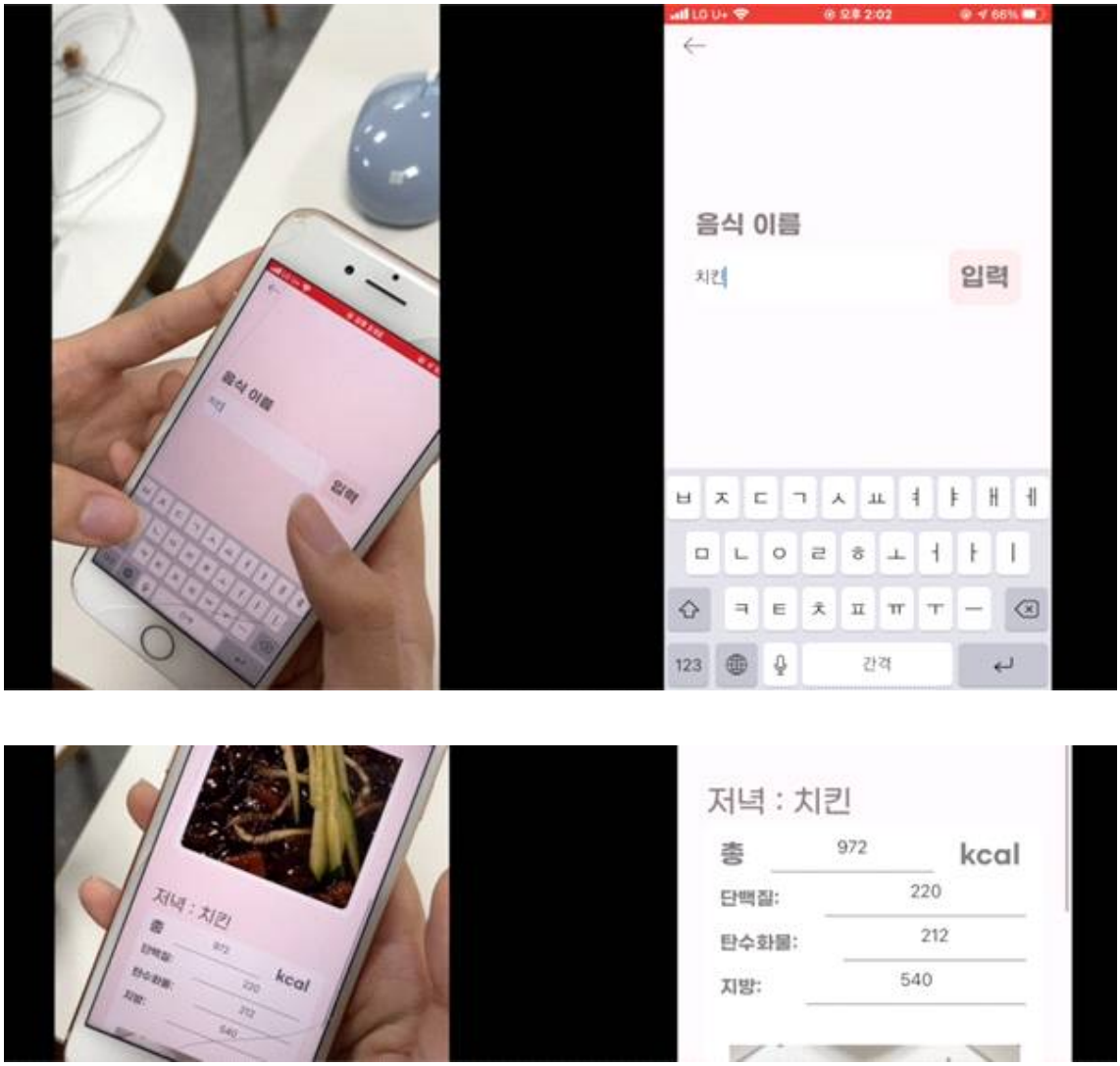

그림 4.12 : 기능 3 사진

이미지 인식을 잘못하였다고 가정하고, 사용자가 직접 정보를 입력하여 칼로리를 입력 받는 모습이다.

Ⅴ. 결 론

본 논문에서는 이미지 인식에 많이 쓰이는 CNN 알고리즘 중 가장 높은 정확도를 보이는 EfficientNet 알고리즘을 이용하여 음식 이미지 분류 모델을 구현하였고, 이 모델을 이용하여 식단 다이어리 어플리케이션을 제작하였다. 비교적 최신의 알고리즘인 EfficientNet 을 사용함으로서 비교적 높은 정확도를 얻을 수 있었으며 epoch와 running rate 를 직접 변경하며 트레이닝을 시켜 모델 간 정확도를 구별할 수 있었다. 또한 핵심 기능들을 직접 어플리케이션으로 구현해 봄으로써, 이론적인 내용뿐만 아니라 실제적으로 pytorch와 tensorflow로 구현된 모델들이 어떻게 적용되는지 알 수 있었다.

향후 더 개발을 진행한다면 누적된 데이터를 분석하여 맞춤 식단 추천서비스와 같은 다양한 서비스를 제공할 수 있을 것이다.

참고문헌

[1] Mingxing Tan, “EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks, *International Conference on Machine Learning*, vol. 5, September 2020.

[2] Weiqing Min, "A Survey on Food Computing", *ACM Computing Serveys*, vol. 52, no. 5, September 2020.

[3] Lei Zhou, "Application of Deep Learning in Food : A Revier", September 2019.

부 록

부록 1. EfficientNet training Source Code

```
#EfficientNet traing model
def train_model(model, criterion, optimizer, scheduler,
num_epochs=25):
    since = time.time()
    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0
    train_loss, train_acc, valid_loss, valid_acc = [], [], [], []

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)
        # Each epoch has a training and validation phase
        for phase in ['train', 'valid']:
            if phase == 'train':
                model.train()  # Set model to training mode
            else:
                model.eval()   # Set model to evaluate mode
            running_loss, running_corrects, num_cnt = 0.0, 0, 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.to(device)
                labels = labels.to(device)
```

```
                # zero the parameter gradients
                optimizer.zero_grad()
                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)
                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()
                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
                num_cnt += len(labels)
            if phase == 'train':
                scheduler.step()

            epoch_loss = float(running_loss / num_cnt)
            epoch_acc  = float((running_corrects.double() /
num_cnt).cpu()*100)

            if phase == 'train':
                train_loss.append(epoch_loss)
                train_acc.append(epoch_acc)
            else:
                valid_loss.append(epoch_loss)
                valid_acc.append(epoch_acc)
            print('{} Loss: {:.2f} Acc: {:.1f}'.format(phase,
epoch_loss, epoch_acc))
```

```
# deep copy the model
            if phase == 'valid' and epoch_acc > best_acc:
                best_idx = epoch
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())
                print('==> best model saved - %d /
%.1f'%(best_idx, best_acc))
        writer.add_scalar('epoch acc', epoch_acc, epoch)
        writer.add_scalar('epoch_loss', epoch_loss, epoch)
        writer.close()
    time_elapsed = time.time() - since
    print('Training complete in {:.0f}m {:.0f}s'.format(time_elapsed //
60, time_elapsed % 60))
    print('Best valid Acc: %d - %.1f' %(best_idx, best_acc))
    # load best model weights
    model.load_state_dict(best_model_wts)
    torch.save(model.state_dict(), 'president_model.pt')
    print('model saved')
    return model, best_idx, best_acc, train_loss, train_acc, valid_loss,
valid_acc

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)
criterion = nn.CrossEntropyLoss()
optimizer_ft = optim.SGD(model.parameters(),
                         lr = 0.05,
                         momentum=0.9,
                         weight_decay=1e-4)
lmbda = lambda epoch: 0.98739
exp_lr_scheduler = optim.lr_scheduler.MultiplicativeLR(optimizer_ft,
lr_lambda=lmbda)
```

```
import warnings
warnings.filterwarnings(action='ignore')
model, best_idx, best_acc, train_loss, train_acc, valid_loss, valid_acc
= train_model(model, criterion, optimizer_ft, exp_lr_scheduler,
num_epochs=200)
```

부록 2. EfficientNet predict Source Code

```
#predict
from PIL import Image
import numpy as np
import cv2
def predict(model,image,device,encoder,transforms =
None,inv_normalize = None):
    model = torch.load('./model_180_0.05_300.h5')
    #model.eval()
    if(isinstance(image,np.ndarray)):
      image = Image.fromarray(image)
    if(transforms!=None):
        image = transforms(image)
    data = image.expand(1,-1,-1,-1)
    data = data.type(torch.FloatTensor).to(device)
    sm = nn.Softmax(dim = 1)
    output = model(data)
    output = sm(output)
    _, preds = torch.max(output, 1)
    img_plot(image,inv_normalize)
    prediction_bar(output,encoder)
    return preds
```

```
def prediction_bar(output,encoder):
    output = output.cpu().detach().numpy()
    a = output.argsort()
    a = a[0]

    size = len(a)
    if(size>5):
        a = np.flip(a[-5:])
    else:
        a = np.flip(a[-1*size:])
    prediction = list()
    clas = list()
    for i in a:
      prediction.append(float(output[:,i]*100))
      clas.append(str(i))
    for i in a:
        print('Class: {} , confidence:
{}'.format(encoder[int(i)],float(output[:,i]*100)))
    # plt.bar(clas,prediction)
    # plt.title("Confidence score bar graph")
    # plt.xlabel("Confidence score")
    # plt.ylabel("Class number")
 def img_plot(image,inv_normalize = None):
    if(inv_normalize!=None):
        image = inv_normalize(image)
    image = image.cpu().numpy().transpose(1,2,0)
    plt.imshow(image)
    plt.show()
print('라면----------')
image =
Image.open('/home/cip-server/PythonHome/myenv/predict_test/t3.jpg'
```

```
pred =
predict(classifier,image,device,encoder,test_transforms,inv_normalize)
print('짜장면----------')
image =
Image.open('/home/cip-server/PythonHome/myenv/predict_test/j_t1.jpg')
pred =
predict(classifier,image,device,encoder,test_transforms,inv_normalize)
print('피자----------')
image =
Image.open('/home/cip-server/PythonHome/myenv/predict_test/p_t2.jpg')
pred =
predict(classifier,image,device,encoder,test_transforms,inv_normalize)
print('칼국수(비빔국수)----------')
image =
Image.open('/home/cip-server/PythonHome/myenv/predict_test/k_t.jpg')
pred =
predict(classifier,image,device,encoder,test_transforms,inv_normalize)
```