Apple Sweeteness Prediction Using Vision Transformer

date: 2023/06/19

Author: Yun Ki Noh 218002267 / EunChan Kim 21801017

Demo Video: https://youtu.be/GrVAf8BUc9s

1. Introduction

In this project, we want to help cutomer to select apples when many apples are mixed without information that which apple is more sweet. Customer would want to spend money effectively, so they will spend more time to select apple. In addition to this, although there are several ways to check apple's sugar content(for example, using juice or infrared light), each way has limitation about damage to apple or high cost. To predict apple's sugar content by brix witout damge to apple and expencise cost, we determine using deep learing model.

Goal

By training deep learning model to predict sugar content of apple(Brix), predicting apple's sugar content in real time and print out the result is available. Since there have been no preceding studies on predicting apple sweetness, the train loss was set to be within 2 MSE (Mean Squared Error) losses, and the error between the actual sweetness value and the predicted value was arbitrarily set to be within plus or minus 5%.

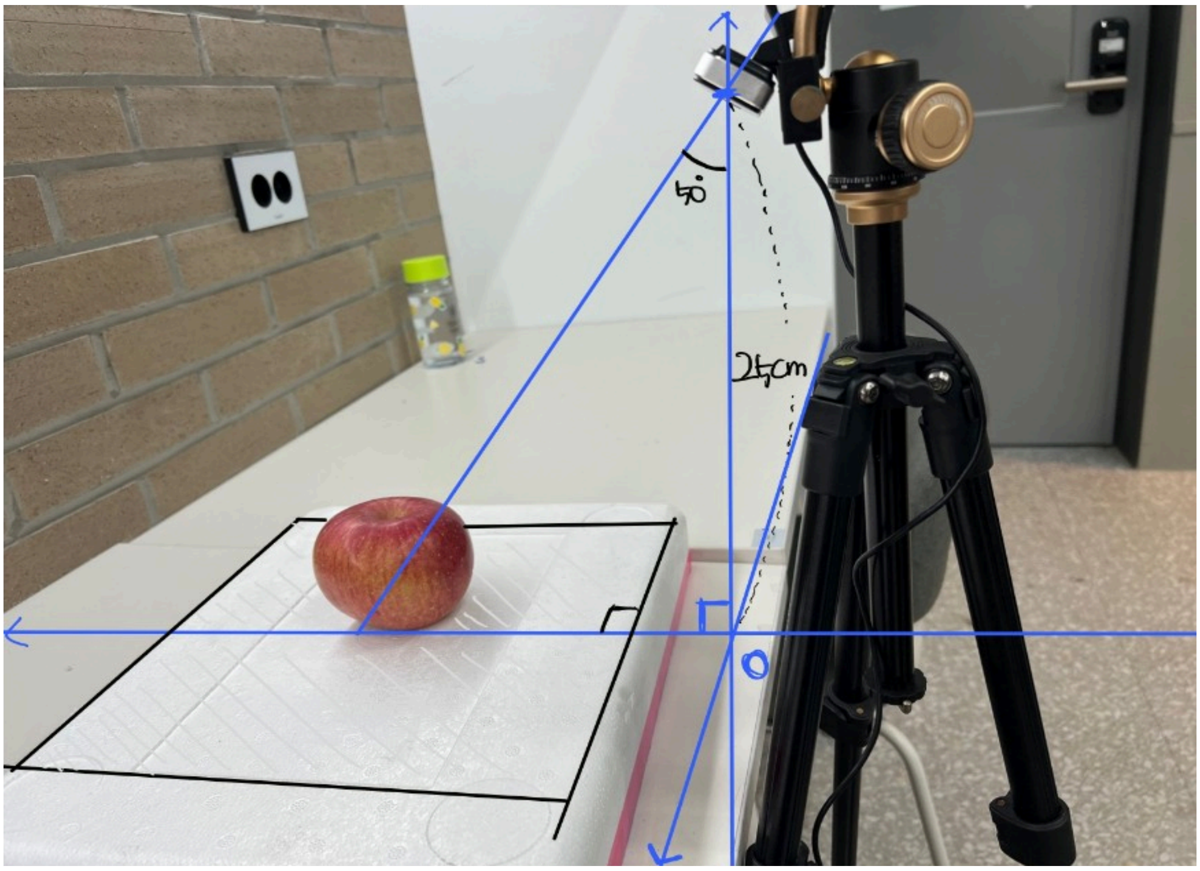
Hardware

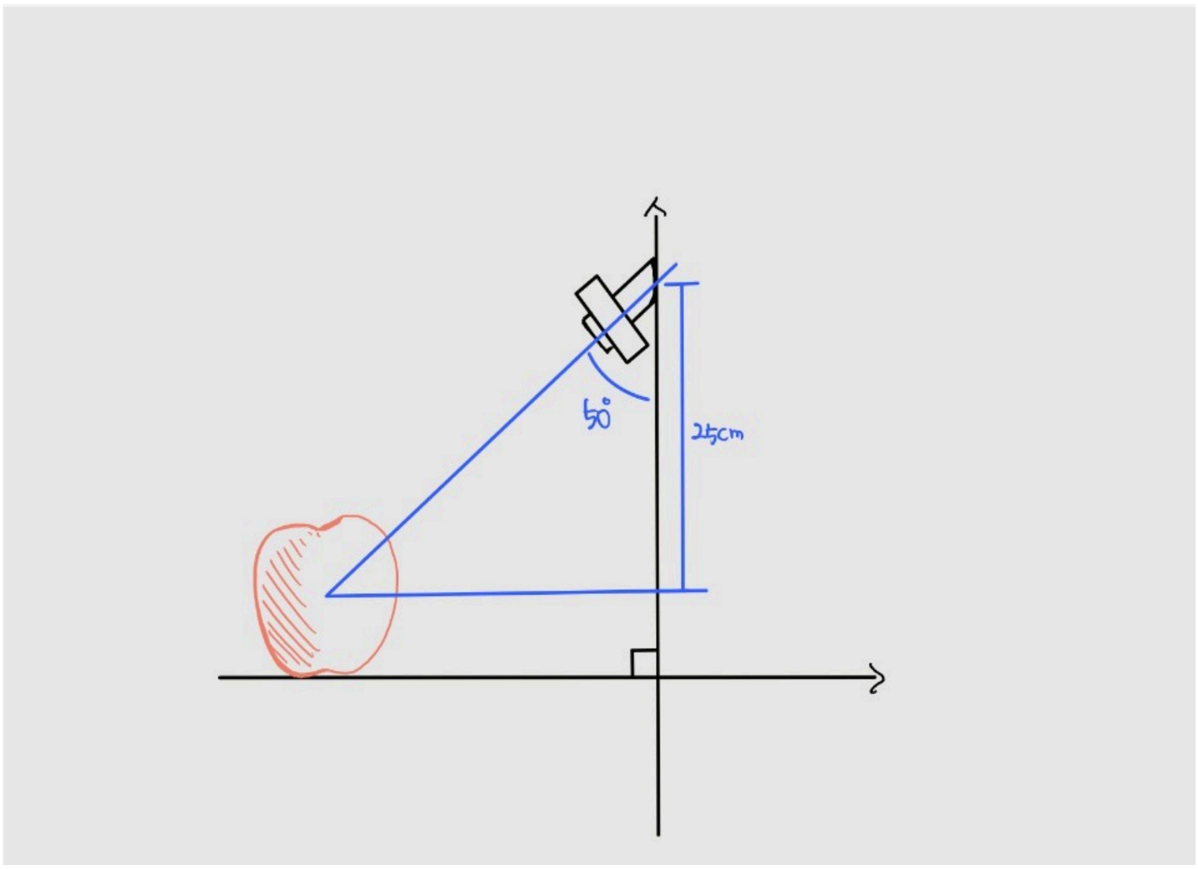
- NVIDIA GeForce RTX 3050
- S604HD VER 1.0
- Server Computer(DSTECH)

Device	Specificaition
GPU	1xAMD EPYC 7742 2.25GHz Upto 3.4GHz
CPU	4EA x NVIDIA A100 80GB (320GB VRAM)

> Table 1. Server Computer

Hardware setting





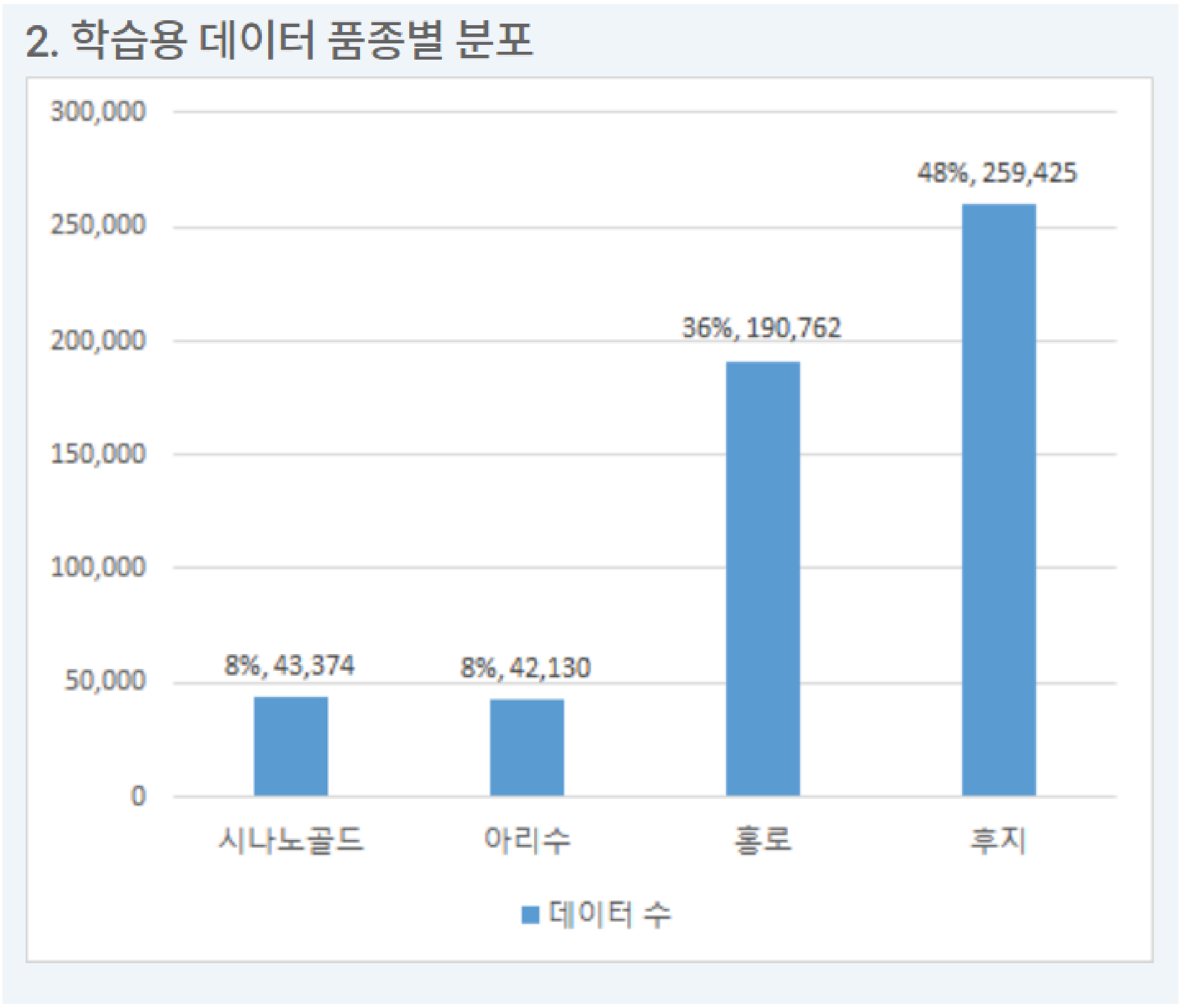
> **Figure 1. Camera's Angle & Distance**

- Camera angle: 50°
- Camera height: 25.0cm

Software

- Window 11
- OpenCV 3.9.13
- NVIDIA Driver Version 526.56
- CUDA 11.7
- PyTorch 1.12.1
- YOLOv8s-seg

2. Dataset



> **Figure 2. Deep Learning Dataset about apple from Jeonbuk**

We used Jeonbuk's apple dataset to train deep learning model. Because Huji apple is most consumed in south korea, we chose 17,000 images and labeling dataset of Huji apple.

Format of dataset

Dataset from Jeonbuk is provided labeling data as json file.

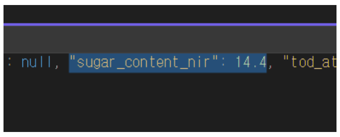
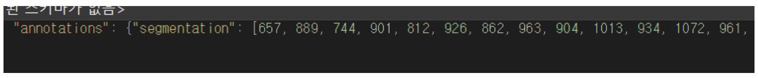
Sugar Content	Segmentation
 `: null, "sugar_content_nir": 14.4, "tod_at`	 `"annotations": {"segmentation": [657, 889, 744, 901, 812, 926, 862, 963, 904, 1013, 934, 1072, 961,`

Table 2. Labeling Data

In the labeling dataset, we used 'segmentation coordination' to extract apple's pixels and 'sugar_content's information' to predict sugar content in Python code.

Dataset link: AI-Hub

3. Tutorial Procedure

Setup & Installation

To train deep learning model in Python, you have to setup several things.

1. Anaconda Virtual Environment

To deal with Python programming in private circumstance, you have to install several necessary programs in your private virtual environment. To create virtual environment, enter this code in anaconda.

```
conda create -n py39 python=3.9.13
```

If you want to activate, enter this code in Anaconda

```
conda activate py39
```

2. Install Libs

Install Numpy, OpenCV, Matplot, Jupyter in your virtual environment

```
conda activate py39
conda install -c anaconda seaborn jupyter
pip install opencv-python
pip install torch torchvision
pip install pillow
pip install numpy
pip install tqdm
pip install opencv-python
pip install matplotlib
pip install timm
pip install ultralytics
```

3. Install Visual Studio Code

Install VS Code by checking this site.

link: [VS Code Downloads]

4-1. Server Computer's CUDA, cuDNN(Option)

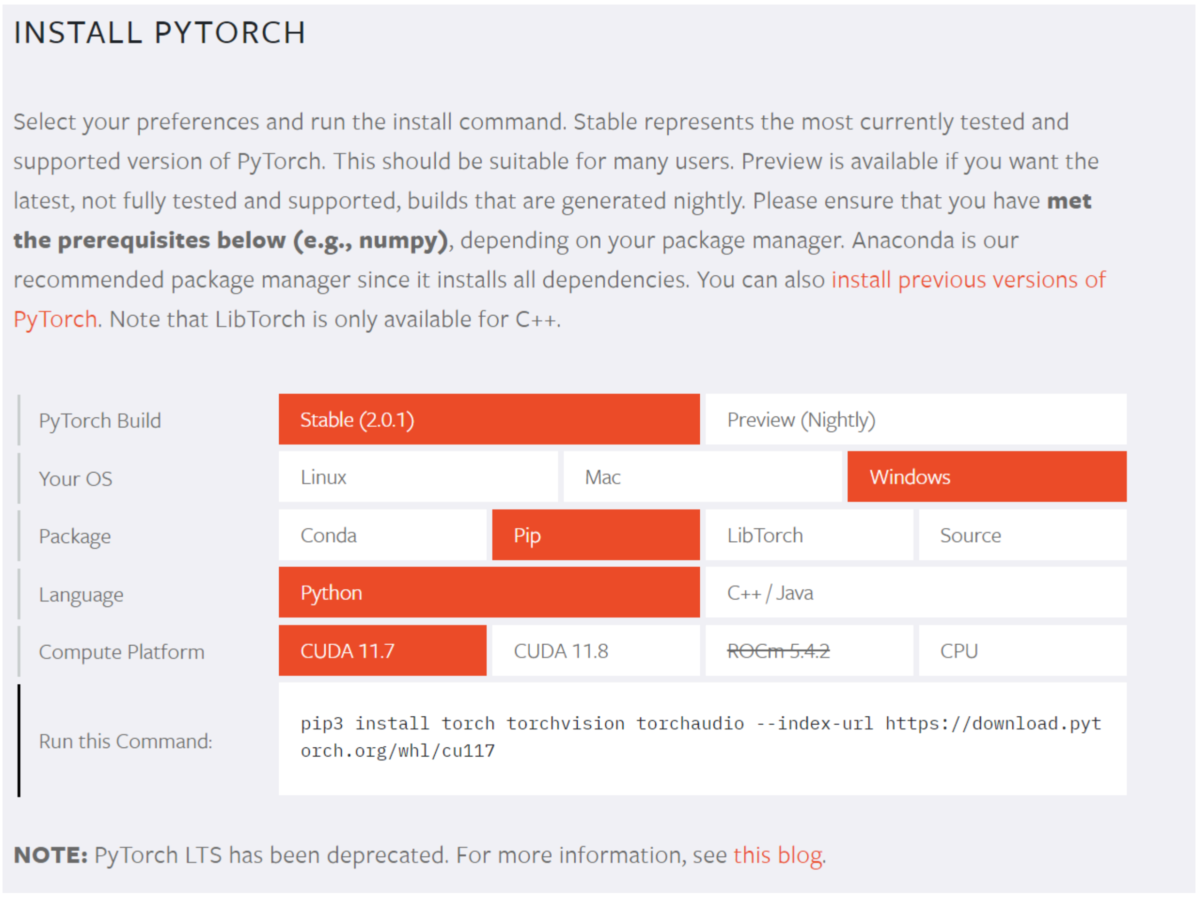
Create private virtual environment and install CUDA and PyTorch

- CUDA 11.3.1
- PyTorch 1.10

4-2. Install GPU Driver, CUDA, cuDNN

Because each driver has proper versions for compatbility, you must select the appropriate version of GPU Driver, CUDA, cuDNN.

1. Check PyTorch & CUDA Version



INSTALL PYTORCH

Select your preferences and run the install command. Stable represents the most currently tested and supported version of PyTorch. This should be suitable for many users. Preview is available if you want the latest, not fully tested and supported, builds that are generated nightly. Please ensure that you have **met the prerequisites below (e.g., numpy)**, depending on your package manager. Anaconda is our recommended package manager since it installs all dependencies. You can also install previous versions of PyTorch. Note that LibTorch is only available for C++.

PyTorch Build	[✓ Stable (2.0.1)]	Preview (Nightly)		
Your OS	Linux	Mac	[✓ Windows]	
Package	Conda	[✓ Pip]	LibTorch	Source
Language	[✓ Python]	C++ / Java		
Compute Platform	[✓ CUDA 11.7]	CUDA 11.8	~~ROCm 5.4.2~~	CPU

Run this Command:

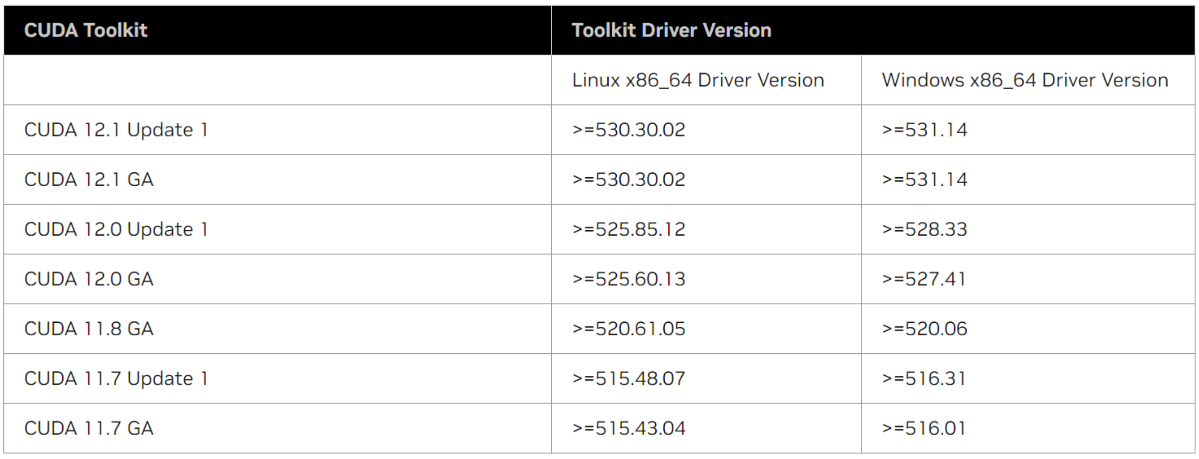
```
pip3 install torch torchvision torchaudio --index-url https://download.pyt
orch.org/whl/cu117
```

NOTE: PyTorch LTS has been deprecated. For more information, see this blog.

Figure 3. Pytorch & CUDA Version

To use PyTorch, we have to use 11.7 or 11.8 version of CUDA.

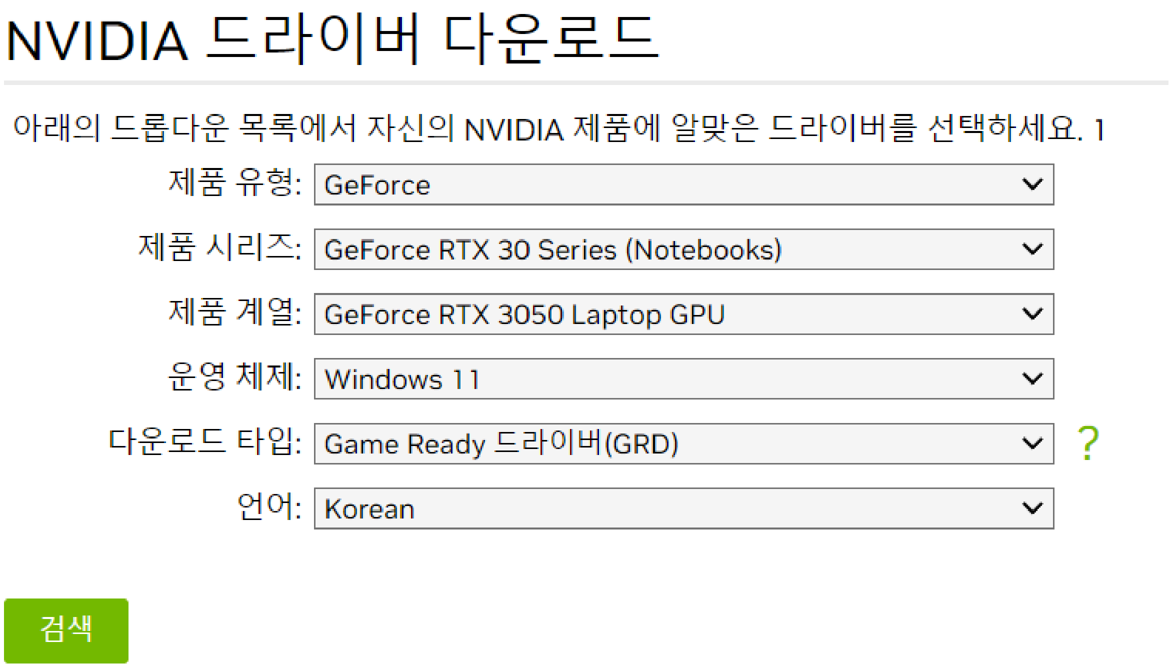
2. Check Proper GPU Version for CUDA 11.7



CUDA Toolkit	Toolkit Driver Version	
	Linux x86_64 Driver Version	Windows x86_64 Driver Version
CUDA 12.1 Update 1	>=530.30.02	>=531.14
CUDA 12.1 GA	>=530.30.02	>=531.14
CUDA 12.0 Update 1	>=525.85.12	>=528.33
CUDA 12.0 GA	>=525.60.13	>=527.41
CUDA 11.8 GA	>=520.61.05	>=520.06
CUDA 11.7 Update 1	>=515.48.07	>=516.31
CUDA 11.7 GA	>=515.43.04	>=516.01

Figure 4. CUDA Toolkit

Install GPU Driver



NVIDIA 드라이버 다운로드

아래의 드롭다운 목록에서 자신의 NVIDIA 제품에 알맞은 드라이버를 선택하세요. 1

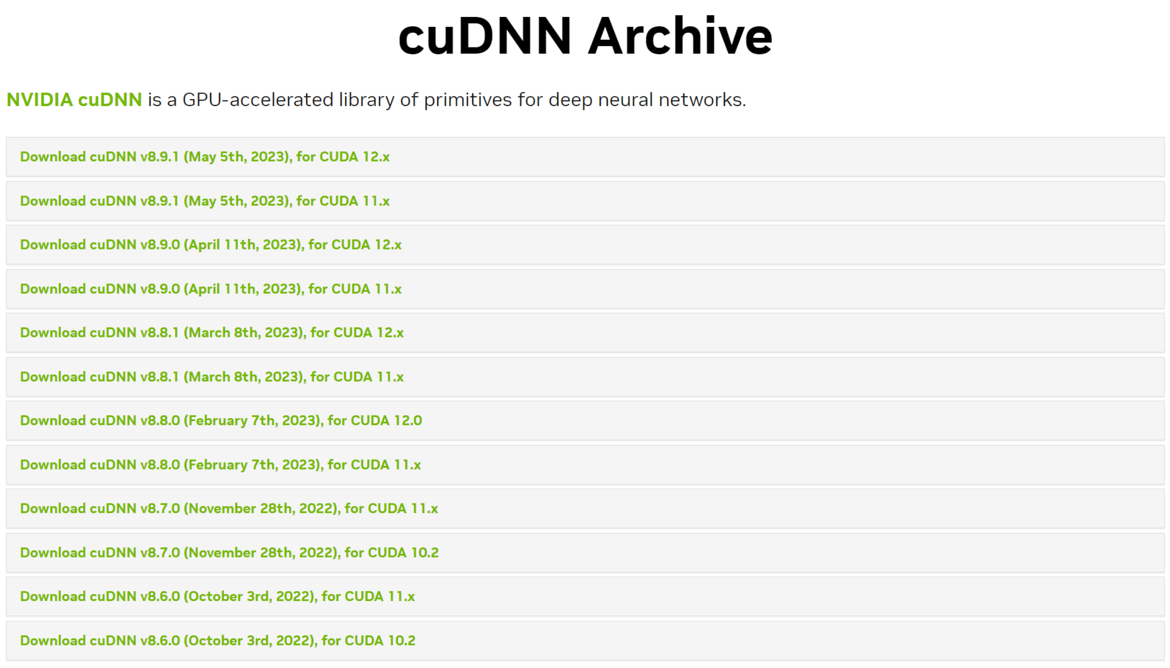
제품 유형:	GeForce
제품 시리즈:	GeForce RTX 30 Series (Notebooks)
제품 계열:	GeForce RTX 3050 Laptop GPU
운영 체제:	Windows 11
다운로드 타입:	Game Ready 드라이버(GRD)
언어:	Korean

Figure 5. GPU Driver

By checking your computer's information, install GPU Driver.

link:GPU Driver

3. Check Proper cuDNN Version for CUDA 11.7


cuDNN Archive

NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks.

Download cuDNN v8.9.1 (May 5th, 2023), for CUDA 12.x
Download cuDNN v8.9.1 (May 5th, 2023), for CUDA 11.x
Download cuDNN v8.9.0 (April 11th, 2023), for CUDA 12.x
Download cuDNN v8.9.0 (April 11th, 2023), for CUDA 11.x
Download cuDNN v8.8.1 (March 8th, 2023), for CUDA 12.x
Download cuDNN v8.8.1 (March 8th, 2023), for CUDA 11.x
Download cuDNN v8.8.0 (February 7th, 2023), for CUDA 12.0
Download cuDNN v8.8.0 (February 7th, 2023), for CUDA 11.x
Download cuDNN v8.7.0 (November 28th, 2022), for CUDA 11.x
Download cuDNN v8.7.0 (November 28th, 2022), for CUDA 10.2
Download cuDNN v8.6.0 (October 3rd, 2022), for CUDA 11.x
Download cuDNN v8.6.0 (October 3rd, 2022), for CUDA 10.2

> **Figure 6. cuDNN version**

link:cuDNN version

4. Install CUDA

Install CUDA 11.7

link:CUDA Toolkit 11.7 Downloads

5. Install PyTorch

Install PyTorch with CUDA 11.7 version

link:PyTorch Downloads

6. Install cuDNN

link:cuDNN Archive

4. Train & Test Process

Train Process

1. Apple Segentation

When measuring the sweetness of an apple, it is necessary to predict the sweetness only within the apple region, rather than the entire image. Therefore, segmentation is essential. After trying various models, we found that the pretrained model provided by Yolov8s-seg performed the best in apple segmentation, so we utilized it. By using the pretrained model, we were able to invest more time in training.

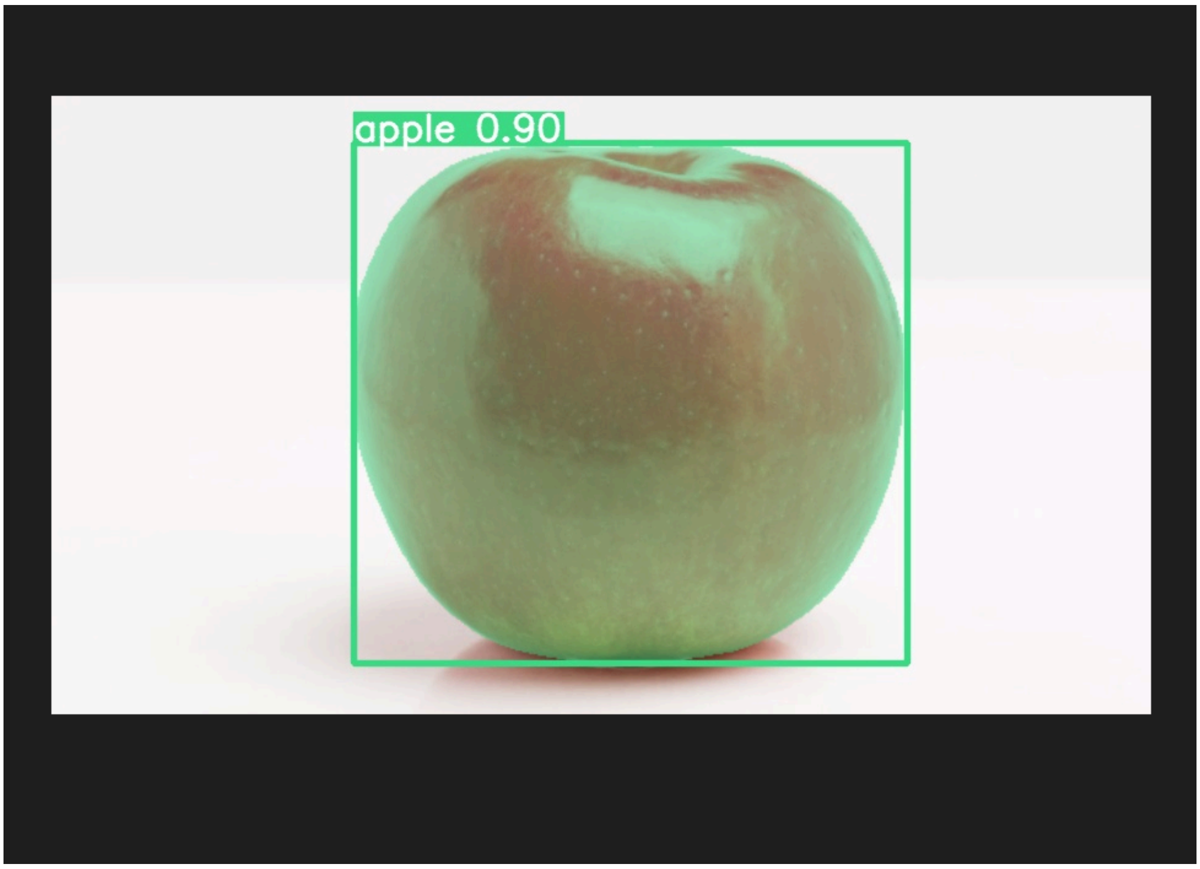


Figure 7. Result of apple segmentation with yolov8-seg

2. Brix Prediction

To predict the sweetness of an apple, the model needs to be trained solely on apple images and corresponding sweetness labels, solving a regression problem for sweetness prediction. Therefore, we determined that using a sophisticated and highly accurate model is appropriate. We experimented with various models such as resNet50, DenseNet, VGG16, Inception v3, among others, but encountered significant loss compared to our expectations.

Figure 8. MES loss comparison of multiple deep learning model

Therefore, we decided to utilize the Vision Transformer model, which has been widely used in deep learning for image processing recently. Specifically, we employed the vit_base_patch16_224 model, which accepts inputs of size 224x224 and utilizes a 16-patch approach.

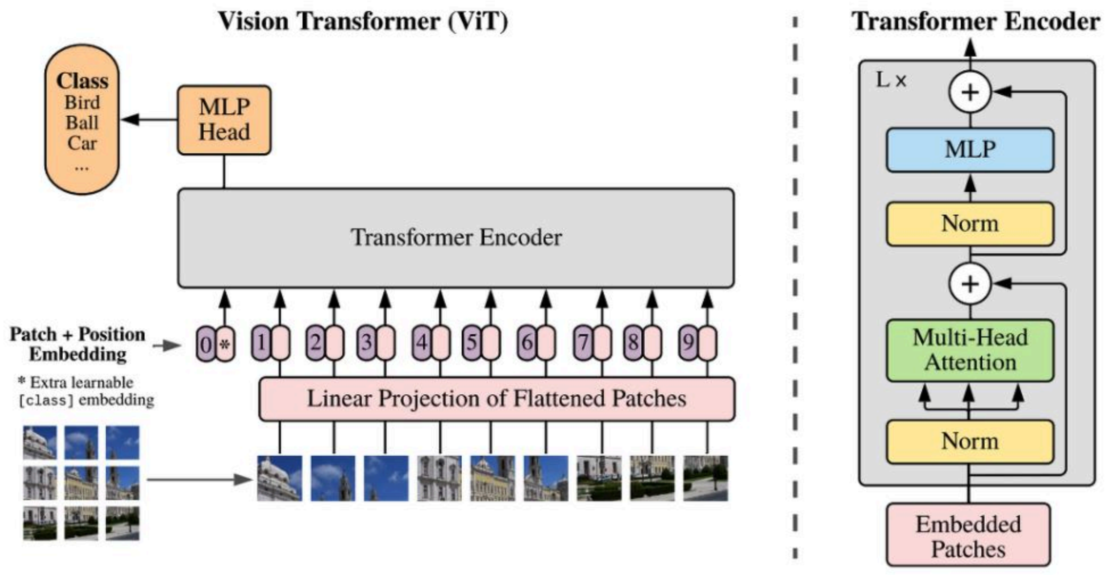


Figure 8. Structure of Vision Transformers (VIT). Referenced by 'Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. ICLR.'

In the provided apple dataset, there was an issue where the segmentation coordinates did not align correctly due to the application of rotation in certain photos. To address this problem, we performed preprocessing by extracting the EXIF information from the photos and adjusting them accordingly if rotation was applied, ensuring proper alignment

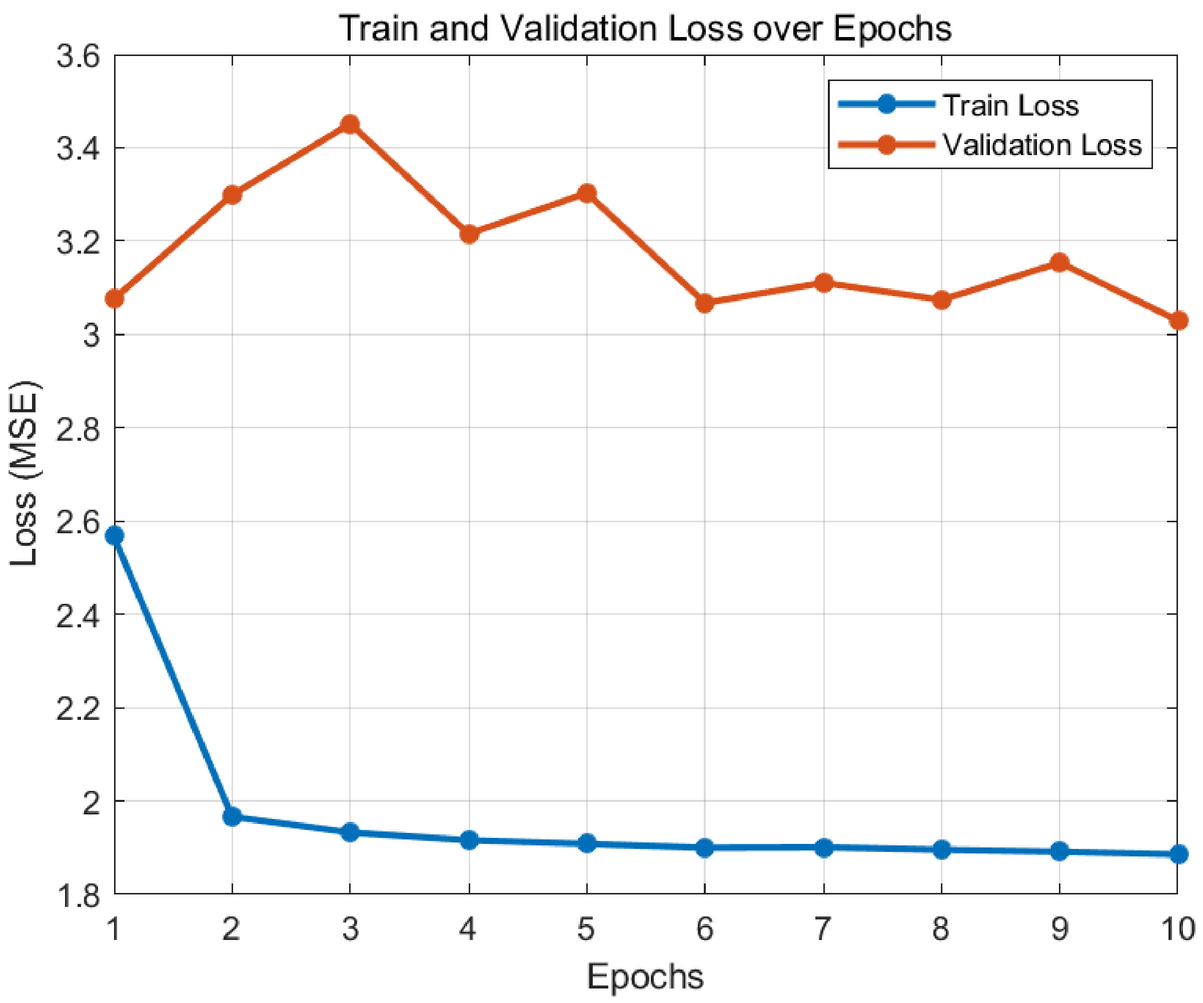
```
def open_and_rotate_image(image_path):
    # if image is rotated correct the image rotation
    image = Image. open(image_path)
    try:
        for orientation in ExifTags.TAGS.keys():
            if ExifTags.TAGS[orientation] == 'Orientation':
                break
        exif = dict(image._getexif().items())
        if exif[orientation] == 3:
            image = image. rotate(180, expand=True)
        elif exif[orientation] == 6:
            image = image. rotate(270, expand=True)
        elif exif[orientation] == 8:
            image = image. rotate(90, expand=True)
    except (AttributeError, KeyError, IndexError):
        # If there's no EXIF ??data or the orientation data isn't set, just
return the original image
        pass
    return image
```

Approximately 170,000 images were used for training, while validation utilized around 17,000 images. Due to the large number of training images, we increased the epoch size. However, this resulted in significantly longer training times. After observing no significant decrease in loss beyond 5 epochs, we set the number of epochs to 10. For loss measurement, we employed the

Mean Squared Error (MSE) loss, commonly used in regression problems. The optimizer used was Adam.

```
num_models = 3
    models_list = []
    for _ in range(num_models):
        # Train using vit_base_patch16_224 model and freeze parameters
        model = timm.create_model('vit_base_patch16_224', pretrained=True)
        for param in model.parameters():
            param.requires_grad = False

        # Update the model's head to output a single value
        model.head = nn.Linear(model.head.in_features, 1)

        # Move the model to the GPU if available
        device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
        model = model.to(device)

        # Add the model to the models list
        models_list.append(model)

    # Use the MSE loss
    criterion = nn.MSELoss()

    # Initialize best validation loss for each model
    best_loss = [float('inf')] * num_models

    for i, model in enumerate(models_list):
        print(f"Training model {i+1}")

        # Initialize Adam optimizer
        optimizer = optim.Adam(model.head.parameters())

        num_epochs = 10

        for i, model in enumerate(models_list):
            print(f"Training model {i+1}")

            # Create lists to store loss per epoch for training and validation
            train_losses = []
            val_losses = []

            for epoch in range(num_epochs):
                model.train()
                running_loss = 0.0
                pbar = tqdm(train_dataloader)

                for inputs, labels in pbar:
                    # Move inputs and labels to the GPU if available
                    inputs, labels = inputs.to(device), labels.to(device).float()
                    # Reshape the labels
                    labels = labels.view(-1, 1)
                    # Zero out the gradients
                    optimizer.zero_grad()
                    # Forward pass
                    outputs = model(inputs)
                    # Calculate the loss
                    loss = criterion(outputs, labels)
                    # Backward pass
                    loss.backward()
                    optimizer.step()

                    running_loss += loss.item() * inputs.size(0)
                    pbar.set_description(f"Train Epoch: {epoch+1}, Loss:
{loss:.4f}")

                # Calculate and print epoch loss for training
                epoch_loss = running_loss / len(train_dataset)
                train_losses.append(epoch_loss)
                print(f'Train Loss: {epoch_loss:.4f}')
```

During the project, it was observed that the set target train loss of less than 2 was met, and it was confirmed that the training proceeded smoothly without any signs of overfitting.



Graph1. Train and Validation Loss Graph

Test Process

Model Download: link

The code was written to utilize the segmentation model and regression model developed in the above process, enabling real-time prediction of apple sweetness. The simple flow is as follows.

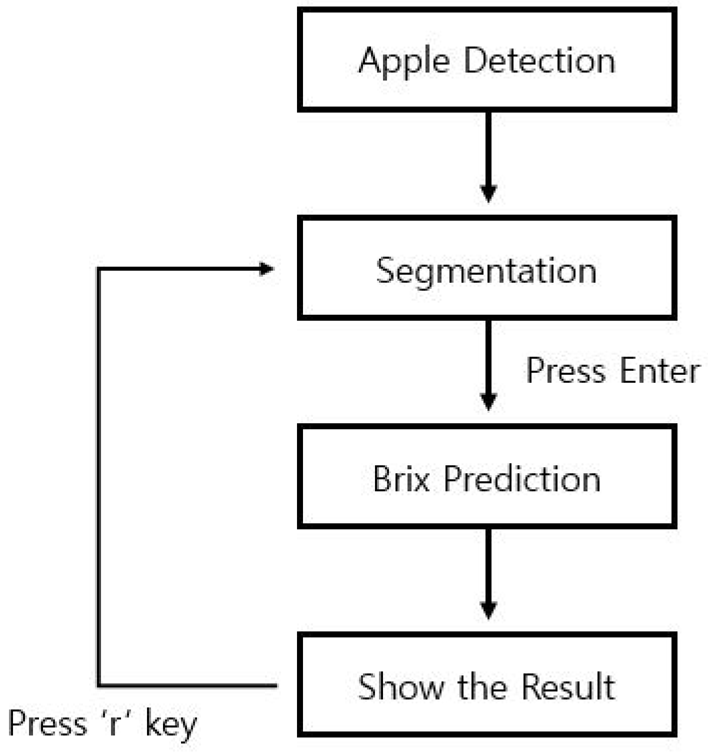


Chart 1. Flow Chart

First, the frame image is resized to a size of 224x224 to make it compatible with the prediction model.

```
# Define transformations for the input
input_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456,
0.406],
                                                            std=[0.229, 0.224,
0.225])])
```

Then, the yolov8-seg model is loaded to perform apple detection. If no apple is detected, the message 'Put the Apple' is displayed.

```
# Load segmentaion model
weight_path = 'yolov8s-seg.pt'

if ret == True: # Run YOLOv8 inference on the frame
        # Change the frame size
        results = model(frame, imgsz=224)
        result = results[0]
        len_result = len(result)
        cv2.namedWindow("video", cv2.WINDOW_AUTOSIZE)

        if len_result == 0:
            # when no object in frame
            cv2.rectangle(frame, (0,0,w,h), (153,255,255), 7)
            cv2.putText(frame, f'Put the Apple', (230, 200), user_font, 0.7,
(153,255,255), 4)
            cv2.putText(frame, f'Put the Apple', (230, 200), user_font, 0.7,
(255,255,255), 2)
```

Once an object is detected and the Enter key is pressed, the segmentation process is initiated. As a result, a mask operation is performed to isolate the data within the apple region. Only the data within the segmented apple area is then utilized for further processing and analysis.

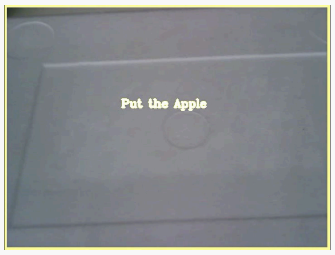
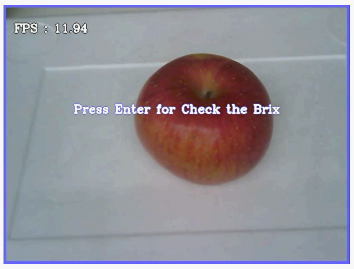
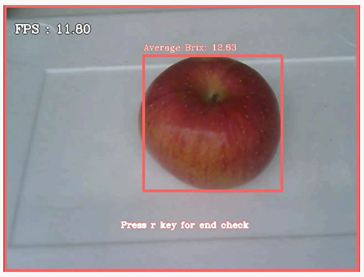
```
# when stop predict Brix
if signal == 0:
    predicted_sugar_contents = []
    cv2.rectangle(frame, (0,0,w,h), (255,102,102), 7)
    cv2.putText(frame, f'Press Enter for Check the Brix', (130, 200), user_font,
0.7, (255,102,102), 4)
    cv2.putText(frame, f'Press Enter for Check the Brix', (130, 200), user_font,
0.7, (255,255,255), 2)

# when predict Brix
elif signal == 1:
    iteration = 0
    cv2.rectangle(frame, (0,0,w,h), (102,102,255), 7)
    cv2.putText(frame, f'Press r key for end check', (210, 400), user_font, 0.5,
(102,102,255), 3)
    cv2.putText(frame, f'Press r key for end check', (210, 400), user_font, 0.5,
(255,255,255), 2)

    # if Apple confidence over 0.7
    if cls == 47 and conf > 0.7:
        iteration += 1

        segment = detection.masks.xy[0].astype(np.int32)
        print("detection.masks",segment)

        color = colors[cls]
        if segment.shape[0] > 0:

            # Use only apple regions for prediction
            mask_base = np.zeros(frame.shape, dtype = np.uint8)
            cv2.fillPoly(mask_base, [segment], (255,255,255))
            img_gray = cv2.cvtColor(mask_base, cv2.COLOR_BGR2GRAY)
            mask2 = cv2.bitwise_and(frame, frame, mask = img_gray)

            # Get bounding box coordinates
            min_x, min_y = np.min(segment, axis=0)
            max_x, max_y = np.max(segment, axis=0)

            # Draw the bounding rectangle
            cv2.rectangle(frame, (min_x, min_y), (max_x, max_y), (102,102,255),
3)  # you can change the color and thickness
```

The received data within the segmented area is used for the prediction of apple sweetness. The predicted sweetness value is then displayed.

```
def predict_sugar_content(_frame):
    # Convert the numpy array frame to a PIL image
    img = Image.fromarray(_frame).convert('RGB')

    # Apply transformations
    img = input_transforms(img)
    img = img.unsqueeze(0)

    # Make prediction
    with torch.no_grad():
        output = model_trans(img)
        predicted = output.item()
        return predicted

model_path = "trans1.pth"
model_trans = timm.create_model('vit_base_patch16_224', pretrained=False)
model_trans.head = nn.Linear(model_trans.head.in_features, 1)
model_trans.load_state_dict(torch.load(model_path))
model_trans.eval()

# In your main loop, after predicting the sugar content, append it to the deque
predicted_sugar_content = round(predict_sugar_content(mask2), 2)
predicted_sugar_contents.append(predicted_sugar_content)

# Calculate the average predicted sugar content from the values in the deque
avg_predicted_sugar_content = sum(predicted_sugar_contents) /
len(predicted_sugar_contents)

cv2.putText(frame, f"Brix: {avg_predicted_sugar_content:.2f}", (min_x+70,
min_y+100), user_font, 0.5, (102,102,255), 2)
cv2.putText(frame, f"Brix: {avg_predicted_sugar_content:.2f}", (min_x+70,
min_y+100), user_font, 0.5, (255,255,255), 1)
```

Finally, the results are displayed on the screen. Pressing the 'r' key takes you back to the previous step of segmentation.

Due to the heavy computational load of the model used for real-time apple sweetness prediction, significant frame drops occur. To mitigate this issue, the prediction is performed only when the apple is placed in position and the Enter key is pressed. To prevent the fluctuation of values with each frame change, the average sweetness value from multiple frames is used. The accumulated sweetness values are continuously added to calculate the average. If the apple is changed, the 'r' key should be pressed to reset the accumulation before proceeding with prediction by pressing the Enter key again.

		 FPS : 11.80 Average Brix: 12.83 Press r key for end check
Figure 9. When No Object in Frame	**Figure 10. When Apple Detected**	**Figure 11. Brix Prediction**

5. Results and Analysis

Results

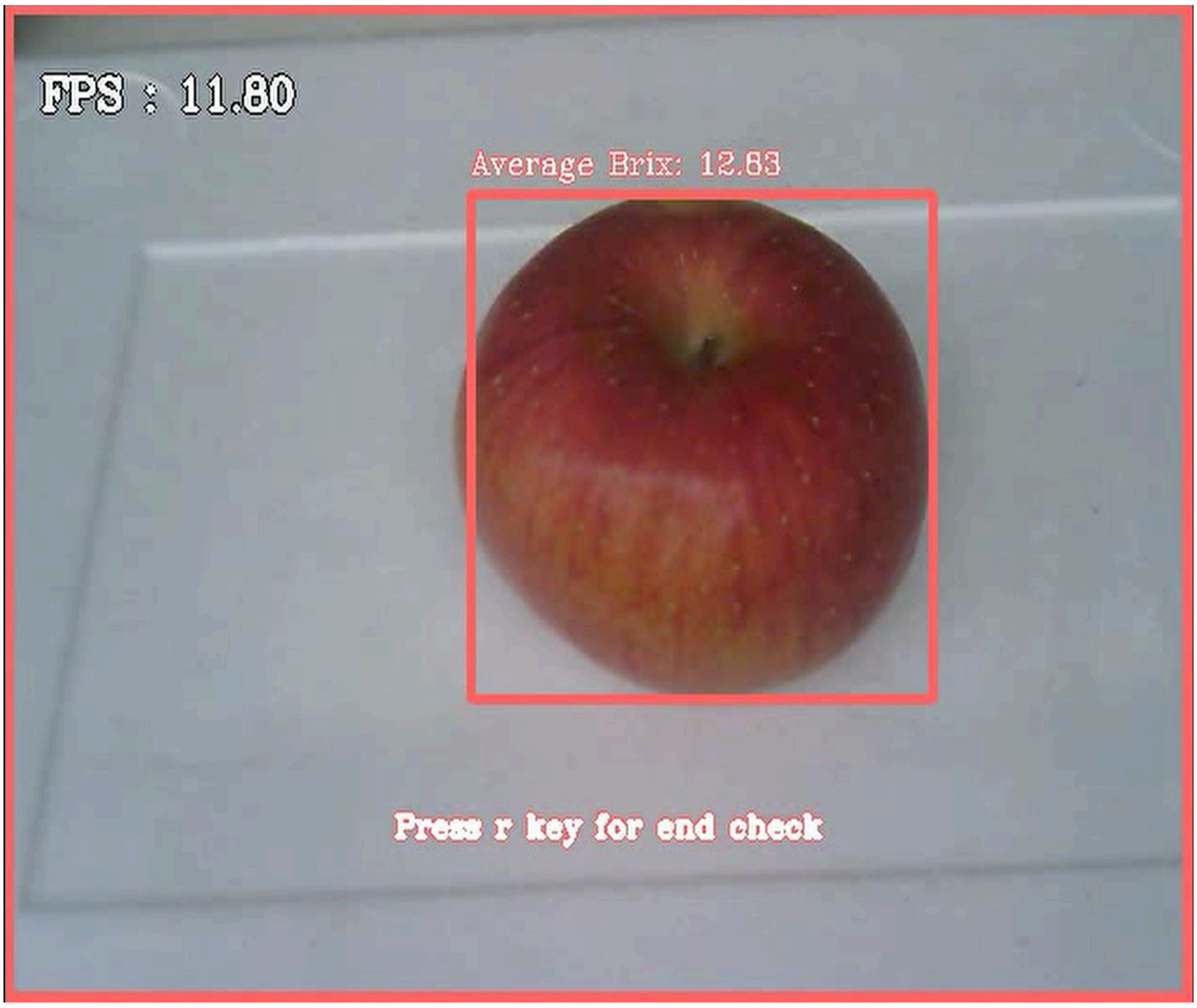


Figure 12. Result

Our project's final result is predicted brix value of apple. Before utilizing this brix value, checking this value is correct is essential.

To check reliability of our trained model, we chose 10 apples and check the error between real value of brix and predicted value of our model. And the result is like this.

	1	2	3	4	5	6	7	8	9	10
Real[brix]	12.5	13.0	13.7	11.0	12.0	10.0	14.2	11.0	13.0	12.0

	1	2	3	4	5	6	7	8	9	10
Prediction[birx]	12.44	12.76	13.15	11.71	12.16	11.53	14.12	11.7	12.88	13.63
Error[%]	0.48	1.85	4.01	6.45	1.33	15.3	0.56	6.36	0.92	13.58

Total[%]
5.08

There were significant errors in a few cases, so we couldn't achieve the goal of an error within 5%.

> **Table 3. Evaluation Error**

	Train Error[MSE]	Validation Error[MSE]	Evaluation Error[%]
Goal	Under 2	No Overfitting	Under 5%
Result	1.89	3.02	5.08
Achievement	O	O	X

> **Table 4. Result**

In the process of training model, we can check train loss under 2 MSE loss and this model is not overfitted by checking validation error. However, we couldn't achieve evaluation goal.

Analysis

Light condition

Upon examining the data used for training, it was observed that more than 70% of the images were captured in an orchard. This indicates that the photos were taken under sufficient lighting conditions, implying that evaluating the model trained indoors may introduce errors. To mitigate this, it is expected that increasing the quantity of training data captured under indoor conditions would lead to improved performance.

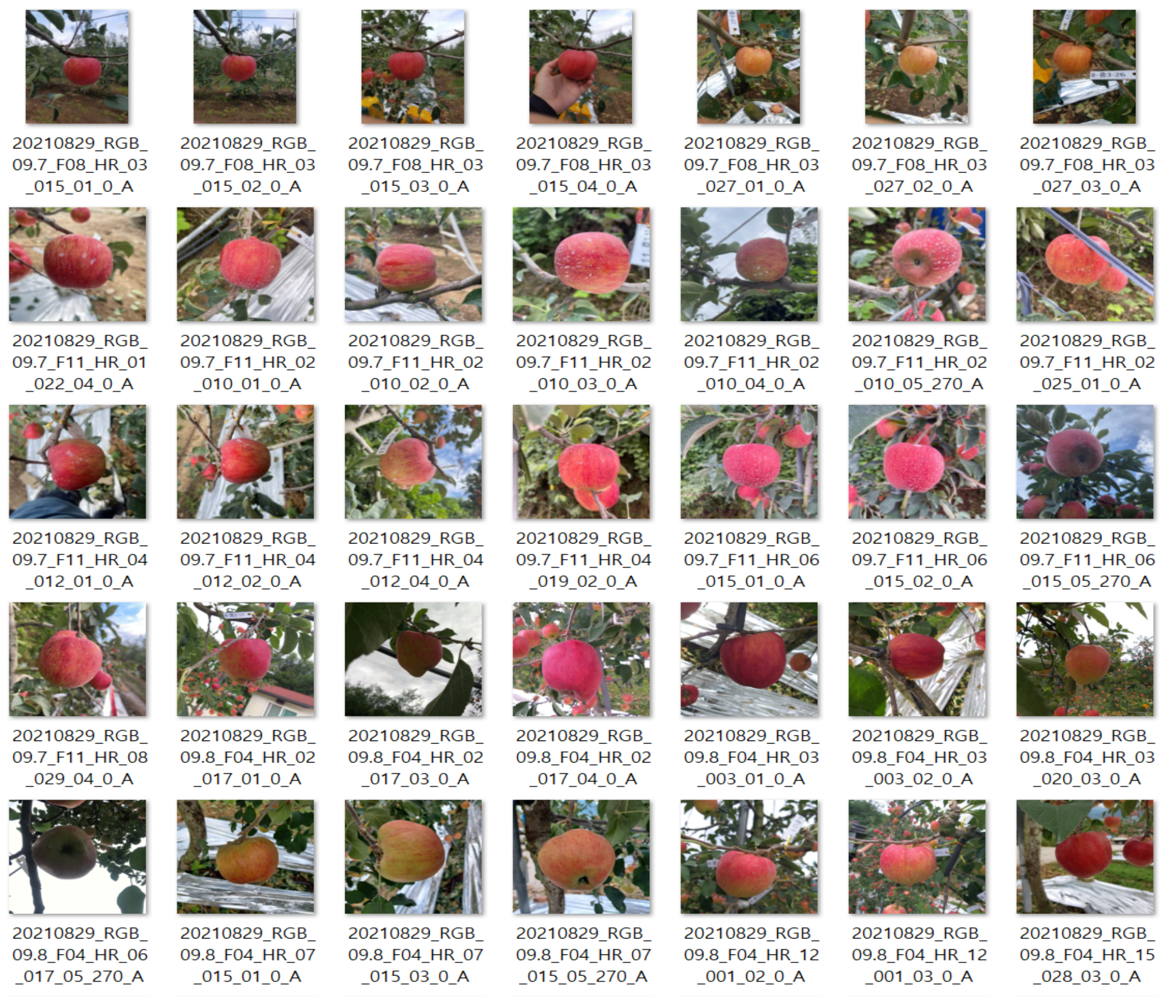

20210829_RGB_09.7_F08_HR_03_015_01_0_A
20210829_RGB_09.7_F08_HR_03_015_02_0_A
20210829_RGB_09.7_F08_HR_03_015_03_0_A
20210829_RGB_09.7_F08_HR_03_015_04_0_A
20210829_RGB_09.7_F08_HR_03_027_01_0_A
20210829_RGB_09.7_F08_HR_03_027_02_0_A
20210829_RGB_09.7_F08_HR_03_027_03_0_A
20210829_RGB_09.7_F11_HR_01_022_04_0_A
20210829_RGB_09.7_F11_HR_02_010_01_0_A
20210829_RGB_09.7_F11_HR_02_010_02_0_A
20210829_RGB_09.7_F11_HR_02_010_03_0_A
20210829_RGB_09.7_F11_HR_02_010_04_0_A
20210829_RGB_09.7_F11_HR_02_010_05_270_A
20210829_RGB_09.7_F11_HR_02_025_01_0_A
20210829_RGB_09.7_F11_HR_04_012_01_0_A
20210829_RGB_09.7_F11_HR_04_012_02_0_A
20210829_RGB_09.7_F11_HR_04_012_04_0_A
20210829_RGB_09.7_F11_HR_04_019_02_0_A
20210829_RGB_09.7_F11_HR_06_015_01_0_A
20210829_RGB_09.7_F11_HR_06_015_02_0_A
20210829_RGB_09.7_F11_HR_06_015_05_270_A
20210829_RGB_09.7_F11_HR_08_029_04_0_A
20210829_RGB_09.8_F04_HR_02_017_01_0_A
20210829_RGB_09.8_F04_HR_02_017_03_0_A
20210829_RGB_09.8_F04_HR_02_017_04_0_A
20210829_RGB_09.8_F04_HR_03_003_01_0_A
20210829_RGB_09.8_F04_HR_03_003_02_0_A
20210829_RGB_09.8_F04_HR_03_020_03_0_A
20210829_RGB_09.8_F04_HR_06_017_05_270_A
20210829_RGB_09.8_F04_HR_07_015_01_0_A
20210829_RGB_09.8_F04_HR_07_015_03_0_A
20210829_RGB_09.8_F04_HR_07_015_05_270_A
20210829_RGB_09.8_F04_HR_12_001_02_0_A
20210829_RGB_09.8_F04_HR_12_001_03_0_A
20210829_RGB_09.8_F04_HR_15_028_03_0_A

Figure 13. Most of Images Taken in Outside

Rotation

Furthermore, it was observed that when the camera and the apple underwent rotation, the sweetness value varied even for the same apple. This could be attributed to the fact that the apple images used for training represented only one side of the apple, potentially introducing errors. It is believed that training with multiple angle images for a given apple-sweetness dataset could help reduce such errors.

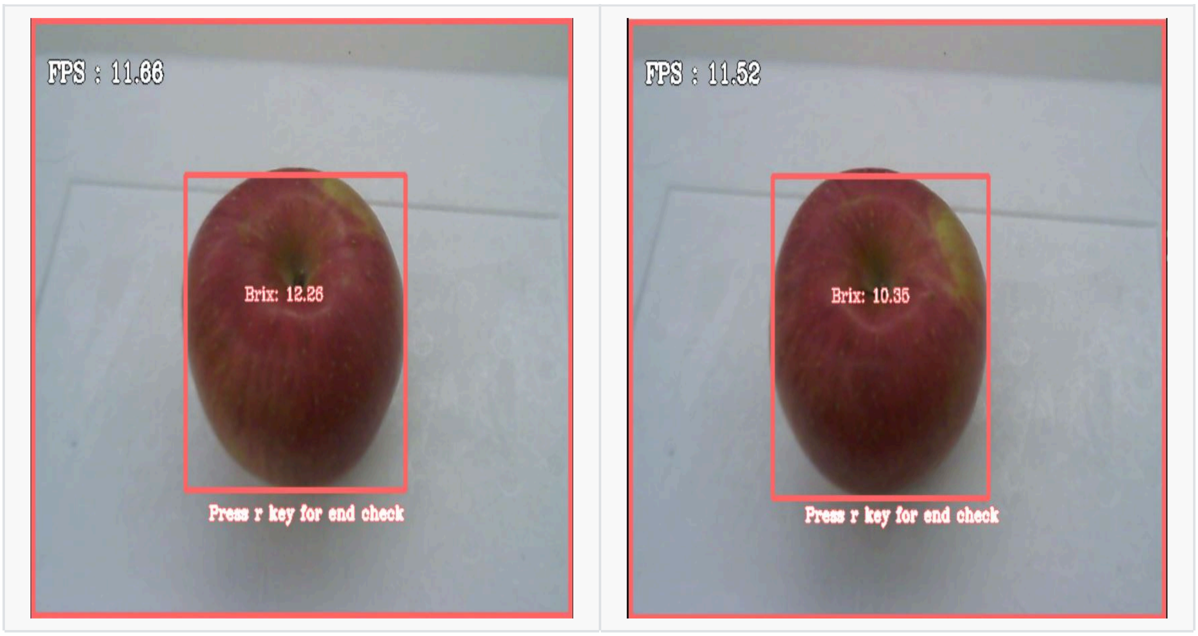


Figure 14. Before Rotation (12.26 Brix)	Figure 15. After Rotation (10.35 Brix)

6. Reference

- 강다영 외 5명. (2021). CNN 을 활용한 수박 당도 예측. ACK 2021.
- 채이한,한지훈. (2021). CNN 기반 이미지 분석을 통한 과일 당도 예측. 세종과학예술영재학교
- Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. ICLR.
- Al-Sammarraie, M.A.J., Gierz, Ł., Przybył, K., Koszela, K., Szychta, M., Brzykcy, J., Baranowska,H.M. (2022). Predicting Fruit's SweetnessUsing Artificial Intelligence—Case Study: Orange. Appl.
- Sangsongfa, A., Am-Dee, N., Meesad P.(2020). Prediction of Pineapple Sweetness from Images Using Convolutional Neural Network. EAI ㄴ

7. Appendix

```
import cv2
import torch
from ultralytics import YOLO
import numpy as np
import time
import torch
from torchvision import transforms, models
from PIL import Image
import torch.nn as nn
import timm
from collections import deque

# Initialize
predicted_sugar_content = 0
signal = 0

# Load segmentaion model
weight_path = 'yolov8s-seg.pt'

model = YOLO(weight_path)

# Cam setting
my_cam_index = 0
cap = cv2.VideoCapture(my_cam_index, cv2.CAP_DSHOW)
cap.set(cv2.CAP_PROP_FPS, 30.0)
cap.set(cv2.CAP_PROP_FOURCC, cv2.VideoWriter.fourcc('M','J','P','G'))
cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 0.75)
print(cap.get(cv2.CAP_PROP_EXPOSURE))

# ----------------------------------------------------------------- #
```

```
width    = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
height   = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
fps      = int(cap.get(cv2.CAP_PROP_FPS))
print('get cam fps: ', fps)
user_font     = cv2.FONT_HERSHEY_COMPLEX

# Video write initialize
fourcc = cv2.VideoWriter_fourcc(*'XVID')
output_filename = 'test.mp4'
frame_size = (width, height)
fps = cap.get(cv2.CAP_PROP_FPS)
out = cv2.VideoWriter(output_filename, fourcc, fps, frame_size)

with open('counting_result.txt', 'w') as f:
    f.write('')

# Define transformations for the input
input_transforms = transforms.Compose([transforms.Resize((224, 224)),
                                       transforms.ToTensor(),
                                       transforms.Normalize(mean=[0.485, 0.456,
0.406],
                                                            std=[0.229, 0.224,
0.225])])

def predict_sugar_content(_frame):
    # Convert the numpy array frame to a PIL image
    img = Image.fromarray(_frame).convert('RGB')

    # Apply transformations
    img = input_transforms(img)
    img = img.unsqueeze(0)

    # Make prediction
    with torch.no_grad():
        output = model_trans(img)
        predicted = output.item()
        return predicted


model_path = "trans1.pth"
model_trans = timm.create_model('vit_base_patch16_224', pretrained=False)
model_trans.head = nn.Linear(model_trans.head.in_features, 1)
model_trans.load_state_dict(torch.load(model_path))
model_trans.eval()


def key_command(_key):


        if _key == ord('s') :   cv2.waitKey()

        elif _key == ord('i'):
        # Get current exposure.
            exposure = cap.get(cv2.CAP_PROP_EXPOSURE)
            # Increase exposure by 1.
            cap.set(cv2.CAP_PROP_EXPOSURE, exposure + 1)
        # Decrease exposure on 'd' key press.
        elif _key == ord('d'):
            # Get current exposure.
            exposure = cap.get(cv2.CAP_PROP_EXPOSURE)
            # Decrease exposure by 1.
            cap.set(cv2.CAP_PROP_EXPOSURE, exposure - 1)

colors = {0: (0, 255, 255),
          47: (0,0,255)  # Red for class 2
          }

# Create a counter
frame_counter = 0

# Initialize the deque to store the last 100 predicted sugar contents
predicted_sugar_contents = deque(maxlen=100)

# Loop through the video frames
while cap.isOpened():

    start_time = time.time()
    prev_time = start_time

    # Read a frame from the video
    ret, frame      = cap.read()

    h, w, c = frame.shape



    if ret == True: # Run YOLOv8 inference on the frame
        # Change the frame size
        results = model(frame, imgsz=224)
        result = results[0]
        len_result = len(result)

        cv2.namedWindow("video", cv2.WINDOW_AUTOSIZE)

        if len_result == 0:
            # when no object in frame
            cv2.rectangle(frame, (0,0,w,h), (153,255,255), 7)
            cv2.putText(frame, f'Put the Apple', (230, 200), user_font, 0.7,
(153,255,255), 4)
            cv2.putText(frame, f'Put the Apple', (230, 200), user_font, 0.7,
(255,255,255), 2)

        for idx in range(len_result):

            detection = result[idx]

            box = detection.boxes.cpu().numpy()[0]
            cls = int(box.cls[0])
            conf = box.conf[0]

            diff_time = time.time() - prev_time
            if diff_time > 0:
                fps = 1 / diff_time

            cv2.putText(frame, f'FPS : {fps:.2f}', (20, 50), user_font, 0.7, (0,
0, 0), 3)
            cv2.putText(frame, f'FPS : {fps:.2f}', (20, 50), user_font, 0.7,
(255, 255, 255), 2)

            # 'Enter' Key: predict Brix
            key = cv2.waitKey(1) & 0xFF
            if key == ord('\r'): signal = 1
            # 'r' Key: Stop predict Brix
            elif key == ord('r'): signal = 0

            # When stop predict Brix
            if signal == 0:
                predicted_sugar_contents = []
                cv2.rectangle(frame, (0,0,w,h), (255,102,102), 7)
                cv2.putText(frame, f'Press Enter for Check the Brix', (130, 200),
user_font, 0.7, (255,102,102), 4)
                cv2.putText(frame, f'Press Enter for Check the Brix', (130, 200),
user_font, 0.7, (255,255,255), 2)

            # When predict Brix
            elif signal == 1:
                iteration = 0
                cv2.rectangle(frame, (0,0,w,h), (102,102,255), 7)
                cv2.putText(frame, f'Press r key for end check', (210, 400),
user_font, 0.5, (102,102,255), 3)
                cv2.putText(frame, f'Press r key for end check', (210, 400),
user_font, 0.5, (255,255,255), 2)

                # if Apple confidence over 0.7
                if cls == 47 and conf > 0.7:
                    iteration += 1

                    segment = detection.masks.xy[0].astype(np.int32)
                    print("detection.masks",segment)

                    color = colors[cls]
                    if segment.shape[0] > 0:

                        # Use only apple regions for prediction
                        mask_base = np.zeros(frame.shape, dtype = np.uint8)
                        cv2.fillPoly(mask_base, [segment], (255,255,255))
                        img_gray = cv2.cvtColor(mask_base, cv2.COLOR_BGR2GRAY)
                        mask2 = cv2.bitwise_and(frame, frame, mask = img_gray)

                        # Get bounding box coordinates
                        min_x, min_y = np.min(segment, axis=0)
                        max_x, max_y = np.max(segment, axis=0)

                        # Draw the bounding rectangle
                        cv2.rectangle(frame, (min_x, min_y), (max_x, max_y),
(102,102,255), 3)  # you can change the color and thickness
```

```
                        # In your main loop, after predicting the sugar content,
append it to the deque
                        predicted_sugar_content =
round(predict_sugar_content(mask2), 2)
                        predicted_sugar_contents.append(predicted_sugar_content)

                        # Calculate the average predicted sugar content from the
values in the deque
                        avg_predicted_sugar_content =
sum(predicted_sugar_contents) / len(predicted_sugar_contents)

                        cv2.putText(frame, f"Brix:
{avg_predicted_sugar_content:.2f}", (min_x+70, min_y+100), user_font, 0.5,
(102,102,255), 2)
                        cv2.putText(frame, f"Brix:
{avg_predicted_sugar_content:.2f}", (min_x+70, min_y+100), user_font, 0.5,
(255,255,255), 1)




        cv2.imshow("video", frame)

        out.write(frame)

        key = cv2.waitKey(1) & 0xFF
        if key == 27   :   break

    else:
        print("Camera is Disconnected ...!")
        break

# Release the video capture object and close the display window
cap.release()
cv2.destroyAllWindows()
```