Data Science
#HW4

Image Classification with Neural Network

산업공학과 201911532 송용재 (2023.10)

Table of Contents

Pytorch, torchvision

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

import torch
import torchvision.models
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data.dataloader import DataLoader
from torchvision.datasets import ImageFolder
from torch.utils.data import random_split
import torchvision.transforms as transforms
from tqdm import tqdm
import torch.optim as optim
```

```python
from torchvision.models import resnet50, ResNet50_Weights
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

```python
dir(torchvision.models)
```

[14] ✓ 0.0s

```
'optical_flow',
'quantization',
'regnet',
'regnet_x_16gf',
'regnet_x_1_6gf',
'regnet_x_32gf',
'regnet_x_3_2gf',
'regnet_x_400mf',
'regnet_x_800mf',
'regnet_x_8gf',
'regnet_y_128gf',
'regnet_y_16gf',
'regnet_y_1_6gf',
'regnet_y_32gf',
'regnet_y_3_2gf',
'regnet_y_400mf',
'regnet_y_800mf',
'regnet_y_8gf',
'resnet',
'resnet101',
'resnet152',
'resnet18',
'resnet34',
'resnet50',
'resnext101_32x8d',
'resnext101_64x4d',
'resnext50_32x4d',
'segmentation',
'shufflenet_v2_x0_5',
```

개발환경


```
device = torch.device('mps') if torch.backends.mps.is_available() else "cpu"
print(device)
```

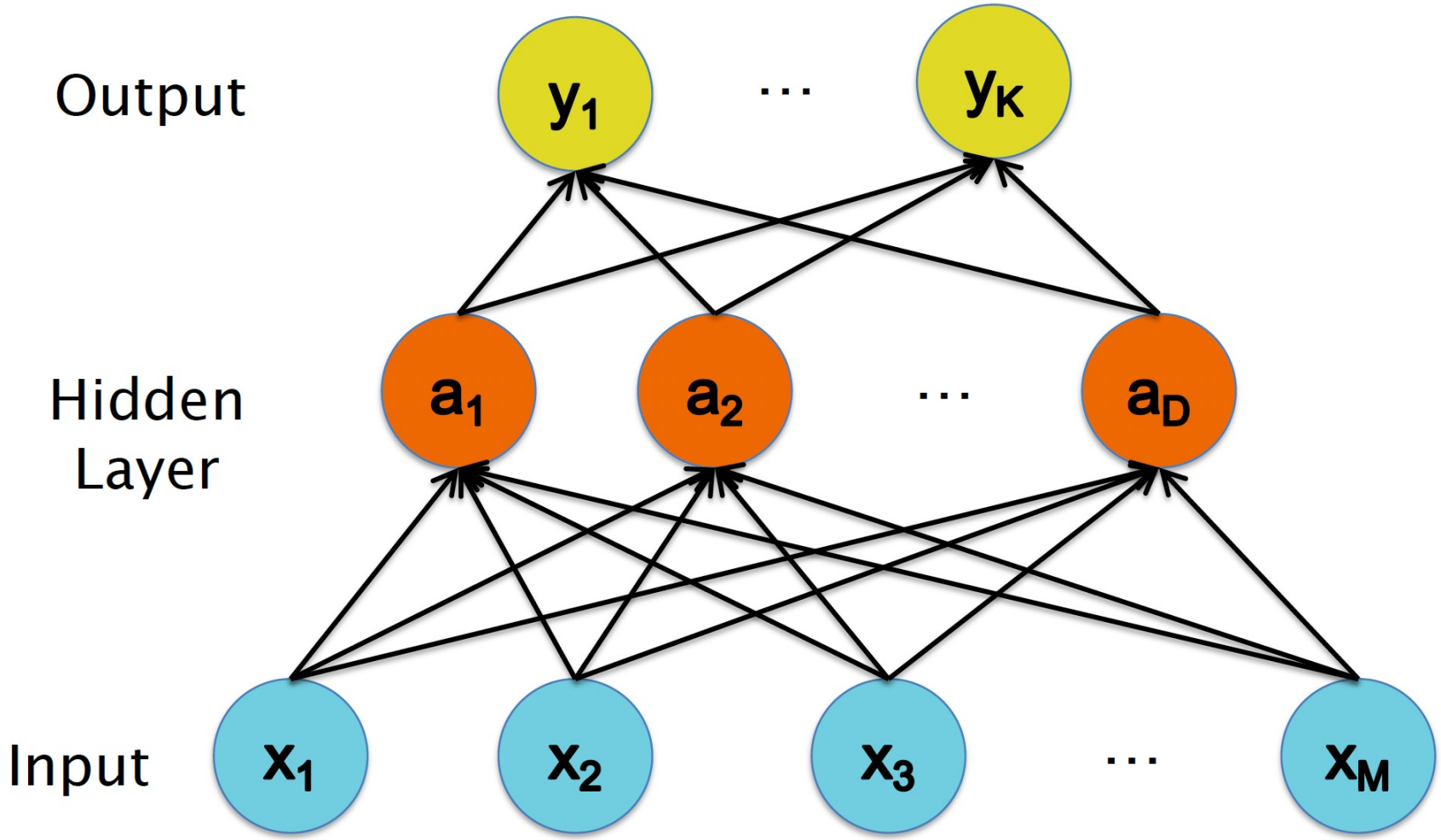
```
... mps
```

MPS BACKEND

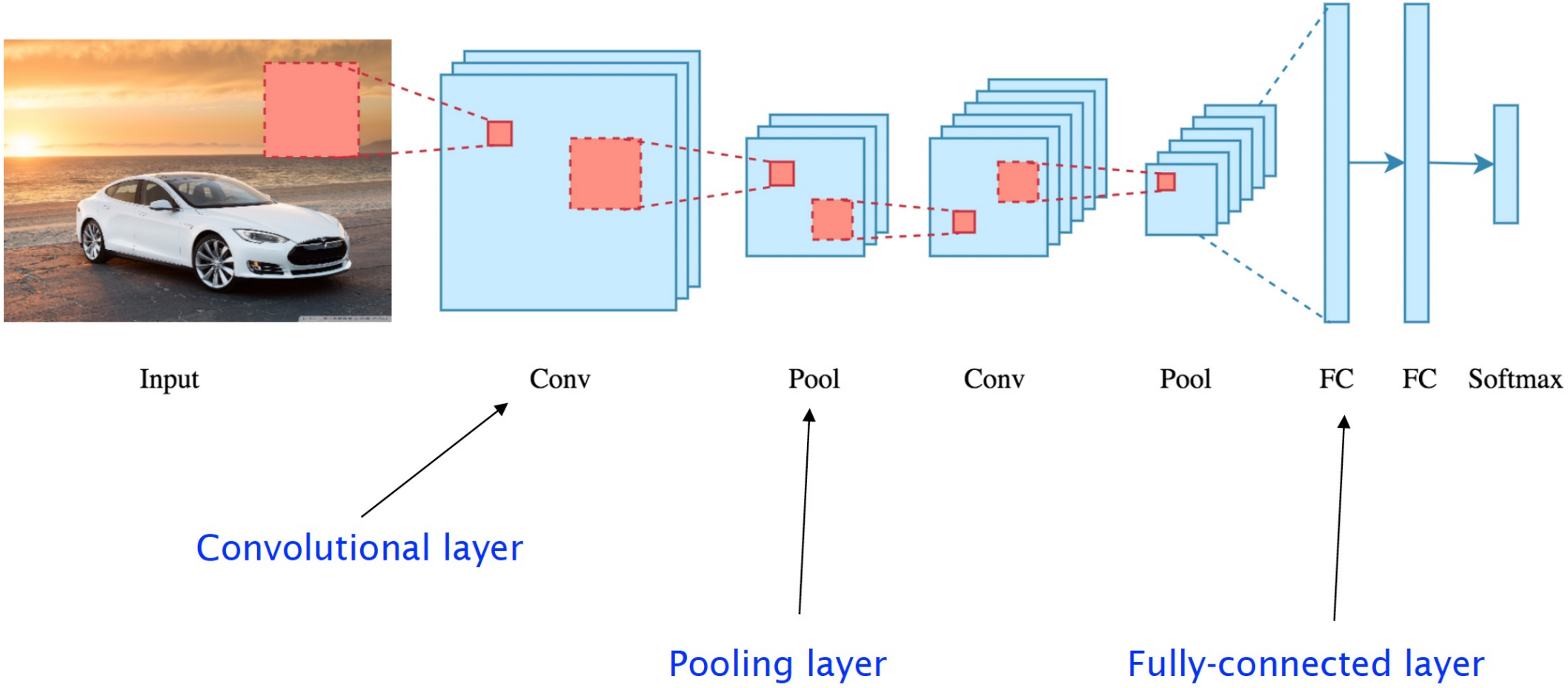
`mps` device enables high-performance training on GPU for MacOS devices with Metal programming framework. It introduces a new device to map Machine Learning computational graphs and primitives on highly efficient Metal Performance Shaders Graph framework and tuned kernels provided by Metal Performance Shaders framework respectively.

The new MPS backend extends the PyTorch ecosystem and provides existing scripts capabilities to setup and run operations on GPU.

Fully Connected Only



Convolution Neural Network



ResNet50

The ResNets developed in [1] are *modularized* architectures that stack building blocks of the same connecting shape. In this paper we call these blocks "*Residual*

Units". The original Residual Unit in [1] performs the following computation:

$$\mathbf{y}_l = h(\mathbf{x}_l) + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l), \tag{1}$$

$$\mathbf{x}_{l+1} = f(\mathbf{y}_l). \tag{2}$$

Here $\mathbf{x}_l$ is the input feature to the l-th Residual Unit. $\mathcal{W}_l = \{\mathrm{W}_{l,k}|_{1\leq k\leq K}\}$ is a set of weights (and biases) associated with the l-th Residual Unit, and K is the number of layers in a Residual Unit (K is 2 or 3 in [1]). $\mathcal{F}$ denotes the residual function, *e.g.*, a stack of two 3×3 convolutional layers in [1]. The function f is the operation after element-wise addition, and in [1] f is ReLU. The function h is set as an identity mapping: $h(\mathbf{x}_l) = \mathbf{x}_l$.[1]

If f is also an identity mapping: $\mathbf{x}_{l+1} \equiv \mathbf{y}_l$, we can put Eqn.(2) into Eqn.(1) and obtain:

$$\mathbf{x}_{l+1} = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l). \tag{3}$$

Recursively ($\mathbf{x}_{l+2} = \mathbf{x}_{l+1} + \mathcal{F}(\mathbf{x}_{l+1}, \mathcal{W}_{l+1}) = \mathbf{x}_l + \mathcal{F}(\mathbf{x}_l, \mathcal{W}_l) + \mathcal{F}(\mathbf{x}_{l+1}, \mathcal{W}_{l+1})$, etc.) we will have:

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}_i, \mathcal{W}_i), \tag{4}$$

ResNet50

Transfer Learning에 매우 효과적으로 사용될 수 있는 구조,

Residual Learning을 기반으로 Skip Connection을 도입하여 Gradient Vanishing 문제 해결의 초석 마련

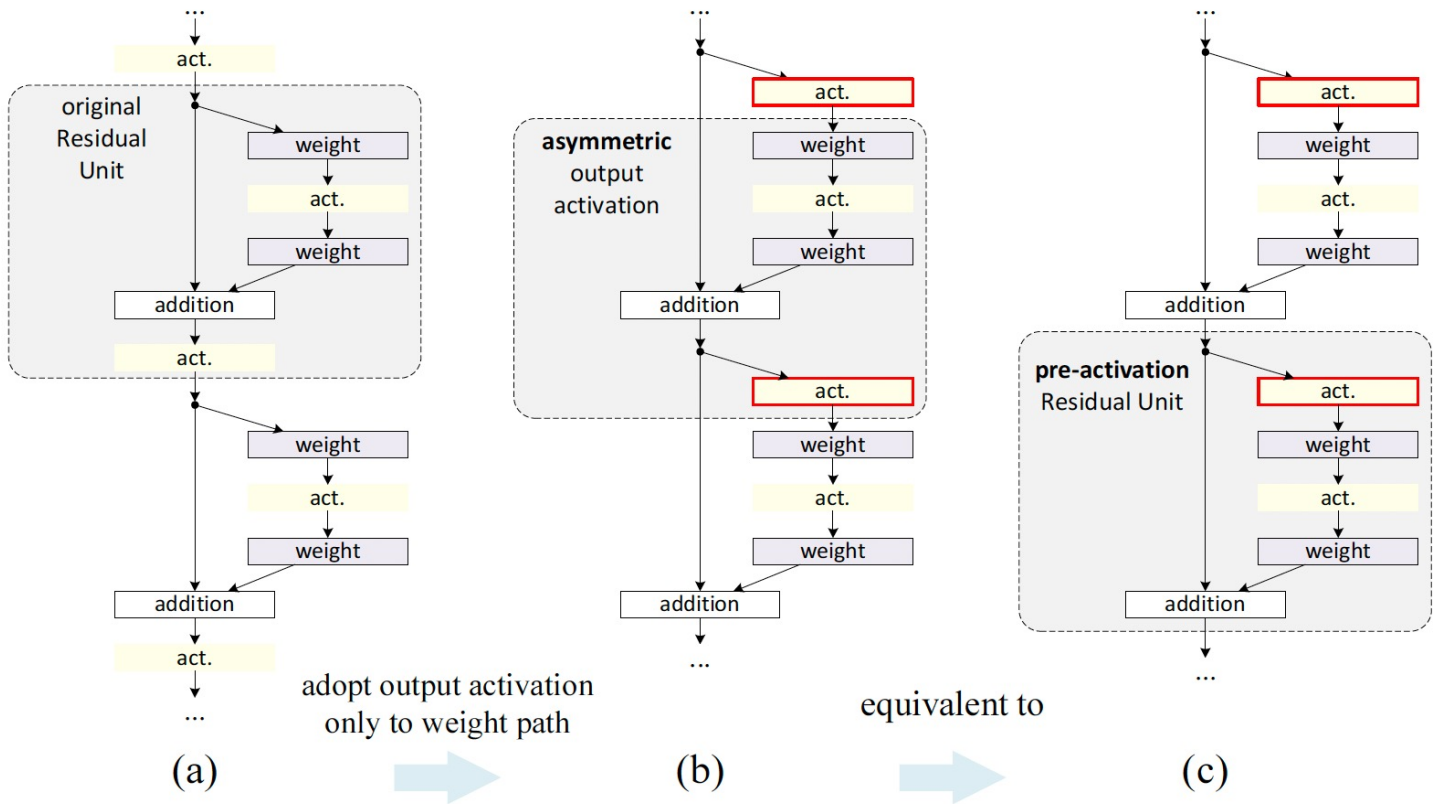


Figure 5. Using asymmetric after-addition activation is equivalent to constructing a *pre-activation* Residual Unit.

main 함수

```python
if __name__ == '__main__':
    device = torch.device('mps') if torch.backends.mps.is_available() else "cpu"
    print(device)

    model1 = ImageClassificationModel_FC()
    model1.to(device)

    model2 = ImageClassificationModel_CNN()
    model2.to(device)

    resnet50(weights=ResNet50_Weights.DEFAULT)
    model3 = building_block_method(num_classes=3, fc_requires_grad=True)
    model3.to(device)

    dataloader = ImageDataLoader()

    trainer1 = ModelTrainer(model1, dataloader.train_loader, dataloader.val_loader, dataloader.test_loader, dataloader.classes, epochs=2)
    trainer1.train(device)
    trainer1.validate(device)
    trainer1.plot_loss()
    trainer1.evaluate(device)

    trainer2 = ModelTrainer(model2, dataloader.train_loader, dataloader.val_loader, dataloader.test_loader, dataloader.classes, epochs=2)
    trainer2.train(device)
    trainer2.validate(device)
    trainer2.plot_loss()
    trainer2.evaluate(device)

    trainer3 = ModelTrainer(model3, dataloader.train_loader, dataloader.val_loader, dataloader.test_loader, dataloader.classes, epochs=2)
    trainer3.train(device)
    trainer3.validate(device)
    trainer3.plot_loss()
    trainer3.evaluate(device)
```

Structure

Structure

Image Data Loader, ‘train : validation : test = 7 : 1 : 2’

```python
class ImageDataLoader:
    def __init__(self):
        transform = transforms.Compose([
        transforms.Resize((300, 300)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])

        root_dir = 'ProjectImages'
        full_train_dataset = ImageFolder(root=root_dir, transform=transform)
        self.classes = full_train_dataset.classes

        total_size = len(full_train_dataset)
        train_size = int(total_size * 0.7)
        val_size = int(total_size * 0.1)
        test_size = total_size - train_size - val_size

        self.train_data, self.val_data, self.test_data = random_split(full_train_dataset, [train_size, val_size, test_size])

        self.train_loader = DataLoader(self.train_data, batch_size=32, shuffle=True, num_workers=2, pin_memory=True)
        self.val_loader = DataLoader(self.val_data, batch_size=32, shuffle=True, num_workers=2, pin_memory=True)
        self.test_loader = DataLoader(self.test_data, batch_size=32, shuffle=True, num_workers=2, pin_memory=True)
```

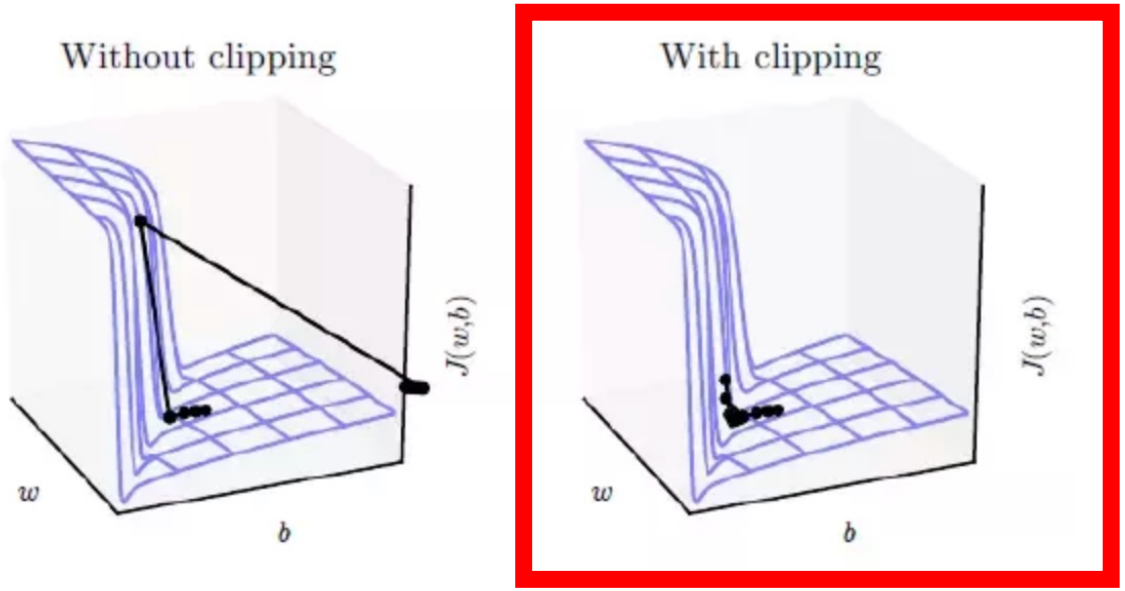
이미지 정규화

파일 호출 및 레이블 분류

Structure

Class. Model Trainer

```python
class ModelTrainer:
    def __init__(self, model, train_loader, val_loader, test_loader, classes, epochs=10, learning_rate=0.001):
        self.model = model
        self.train_loader = train_loader
        self.val_loader = val_loader
        self.test_loader = test_loader
        self.classes = classes
        self.criterion = nn.CrossEntropyLoss()
        self.optimizer = optim.Adam(model.parameters(), lr=learning_rate)

        self.epochs = epochs

        self.train_losses = []
        self.val_losses = []
```



Class_Model_Trainer

Model training with **Progress Bar** (tqdm)

```python
def train(self, device):
    self.model.to(device)
    for epoch in range(self.epochs):
        self.model.train()
        train_loss = 0
        with tqdm(total=len(self.train_loader), desc=f"Epoch {epoch + 1}/{self.epochs}", unit="batch", leave=False) as pbar:
            for features, labels in self.train_loader:
                features, labels = features.to(device), labels.to(device)
                outputs = self.model(features)
                loss = self.criterion(outputs, labels)
                train_loss += loss.item()

                self.optimizer.zero_grad()
                loss.backward()
                self.optimizer.step()

                pbar.set_postfix({'train_loss': f'{loss.item():.4f}'})
                pbar.update(1)

        train_loss /= len(self.train_loader)
        val_loss = self.validate(device)

        self.train_losses.append(train_loss)
        self.val_losses.append(val_loss)

        print(f"Epoch {epoch + 1}/{self.epochs} - Train Loss: {train_loss:.4f}, Validation Loss: {val_loss:.4f}")
```

최적화 및 update 저장

Class_Model_Trainer

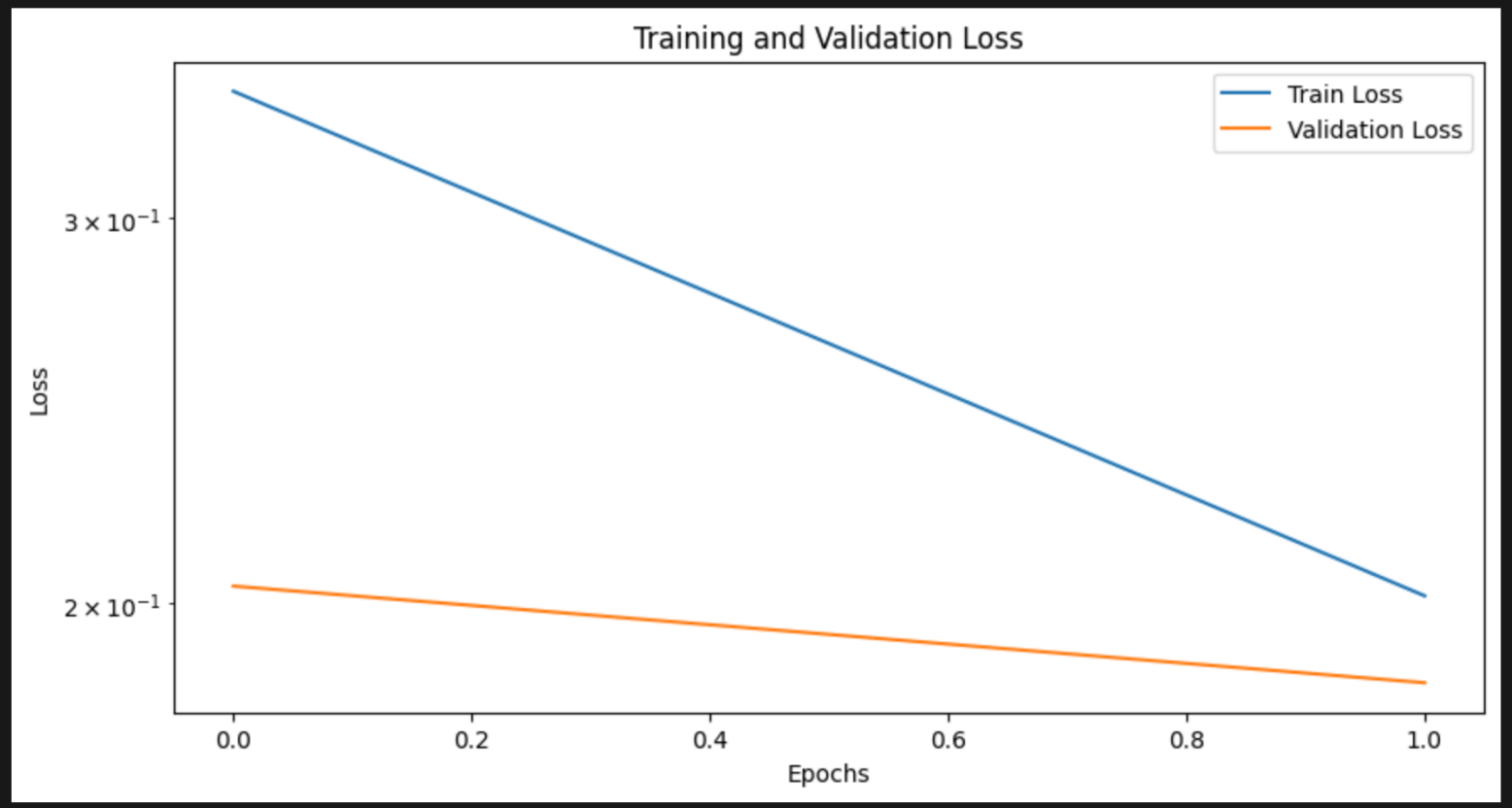
Validate & Plot loss

```python
def validate(self, device):
    self.model.eval()
    val_loss = 0
    with torch.no_grad():
        for features, labels in self.val_loader:
            features, labels = features.to(device), labels.to(device)
            outputs = self.model(features)
            loss = self.criterion(outputs, labels)
            val_loss += loss.item()

    val_loss /= len(self.val_loader)
    return val_loss

def plot_loss(self):
    plt.figure(figsize=(10, 5))
    plt.plot(self.train_losses, label='Train Loss')
    plt.plot(self.val_losses, label='Validation Loss')
    plt.title('Training and Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.yscale('log')
    plt.legend()
    plt.show()
```

Class_Model_Trainer

'Train Loss, Validation Loss & Plot Loss' **Examples**

```
mps

Epoch 1/2 - Train Loss: 0.3430, Validation Loss: 0.2036

Epoch 2/2 - Train Loss: 0.2016, Validation Loss: 0.1839
```



Class_Model_Trainer

Evaluate

Output으로의 분류를 위한 Softmax 함수

$$s\left(x_i\right) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

```python
def evaluate(self, device):
    self.model.eval()
    test_loss = 0
    all_labels = []
    all_preds = []

    total = 0
    correct = 0

    correct_predictions = []
    incorrect_predictions = []


    with torch.no_grad():
        for features, labels in self.test_loader:
            features, labels = features.to(device), labels.to(device)
            outputs = self.model(features.float())
            loss = self.criterion(outputs, labels)
            test_loss += loss.item()

            output_prob = F.softmax(outputs, dim=1).cpu().numpy()

            _, predicted = torch.max(outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_preds.extend(predicted.cpu().numpy())

            for f, l, p, o, prob in zip(features, labels, predicted, outputs, output_prob):
                if l == p and len(correct_predictions) < 10:
                    correct_predictions.append((f, l, p, o, prob))
                elif l != p and len(incorrect_predictions) < 10:
                    incorrect_predictions.append((f, l, p, o, prob))
                if len(correct_predictions) >= 10 and len(incorrect_predictions) >= 10:
                    break

    print("\n----- Correct Predictions -----")
```

Class_Model_Trainer

Evaluate, '맞은 예측 10개 & 틀린 예측 10개 + Accuracy & Confusion Matrix'

```
print("\n----- Correct Predictions -----")
for f, l, p, o, prob in correct_predictions:
    f_permute = f.permute(1, 2, 0).cpu().numpy()

    restored_image = (f_permute * [0.229, 0.224, 0.225]) + [0.485, 0.456, 0.406]
    restored_image = np.clip(restored_image, 0, 1)

    plt.imshow(restored_image, cmap='gray')
    plt.title(f'Predicted: {self.classes[p]}, Actual: {self.classes[l]}')
    plt.show()
    print(f"{o}->{prob}: {self.classes[l]} -> {self.classes[p]}")

print("\n----- Incorrect Predictions -----")
for f, l, p, o, prob in incorrect_predictions:
    f_permute = f.permute(1, 2, 0).cpu().numpy()

    restored_image = (f_permute * [0.229, 0.224, 0.225]) + [0.485, 0.456, 0.406]
    restored_image = np.clip(restored_image, 0, 1)

    plt.imshow(restored_image, cmap='gray')
    plt.title(f'Predicted: {self.classes[p]}, Actual: {self.classes[l]}')
    plt.show()
    print(f"{o}->{prob}: {self.classes[l]} -> {self.classes[p]}")

test_loss /= len(self.test_loader)
print(f"Accuracy = {correct/total: .4f}")

cm = confusion_matrix(all_labels, all_preds)
print(f"Test Loss: {test_loss:.4f}")
print("Confusion Matrix:")
print(cm)
```

정규화 된 이미지를
다시 원 상태로 복구

Fully Connected Only

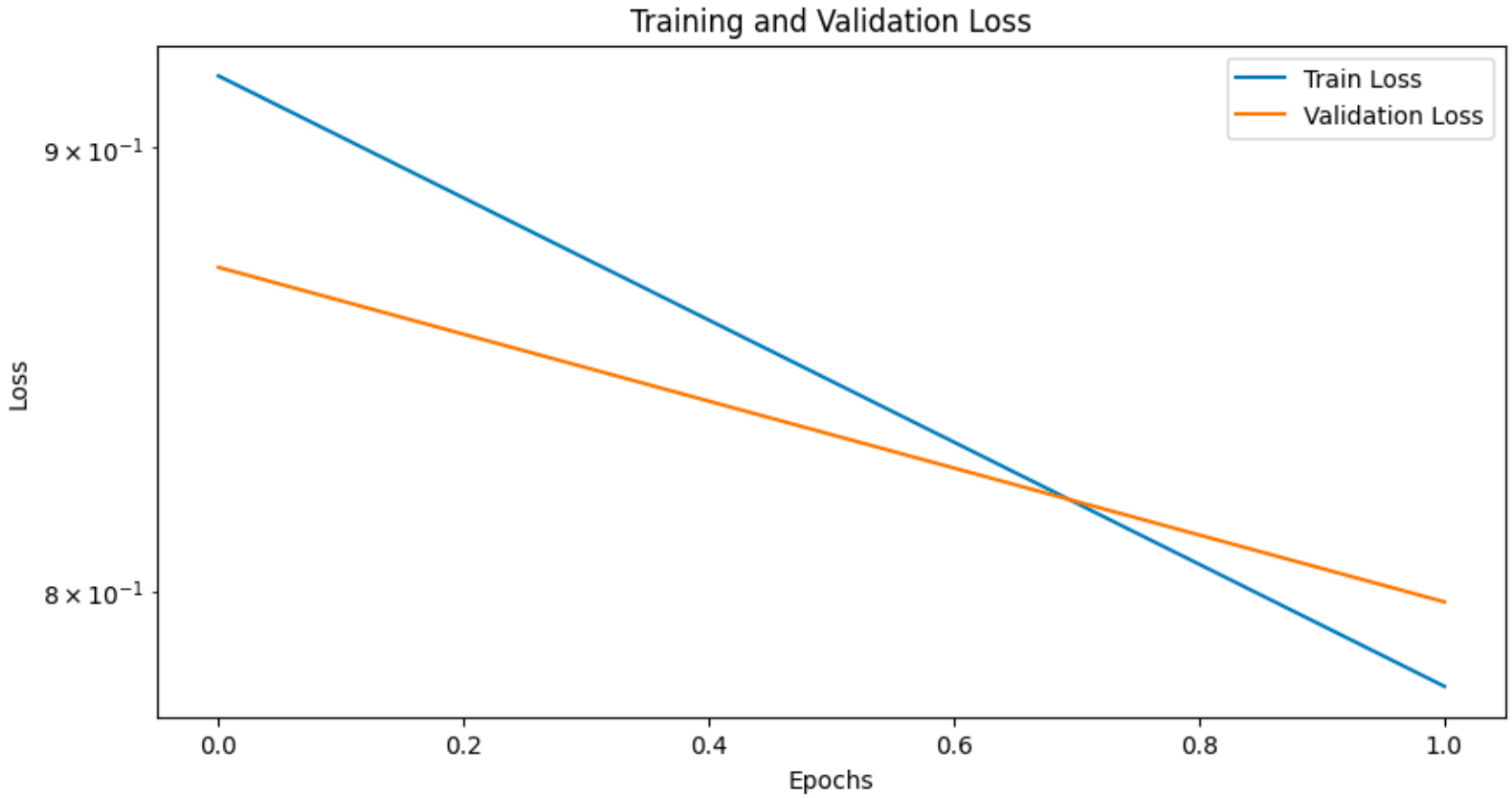
nn.Module

Hidden Layer는 6개, hidden layer 대체로 128 x 128 형태, Dropout(50%)을 통해 Overfitting 방지

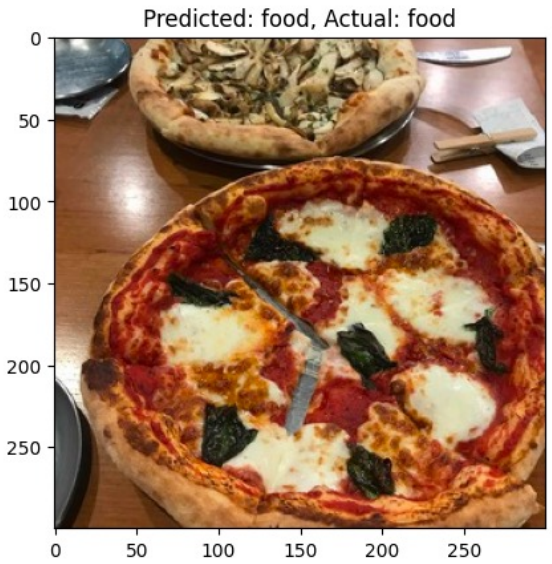
```
class ImageClassificationModel_FC(nn.Module):
    def __init__(self):
        super(ImageClassificationModel_FC, self).__init__()
        self.fc1 = nn.Linear(3 * 300 * 300, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128 ,128)
        self.fc4 = nn.Linear(128 ,128)
        self.fc5 = nn.Linear(128 ,128)
        self.fc6 = nn.Linear(128 ,128)

        self.dropout = nn.Dropout(0.5)

        self.fcm = nn.Linear(128, 3)

    def forward(self, x):
        x = x.reshape(x.size(0), -1)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.relu(self.fc3(x))
        x = torch.relu(self.fc4(x))
        x = torch.relu(self.fc5(x))
        x = torch.relu(self.fc6(x))

        x = self.dropout(x)

        x = self.fcm(x)

        return x
```

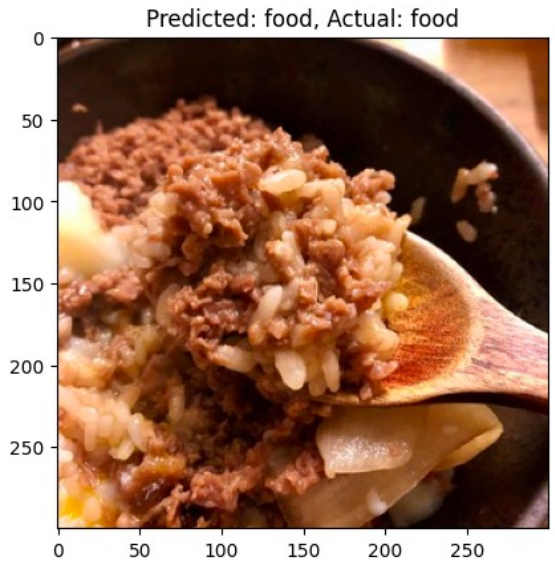
Plot Loss

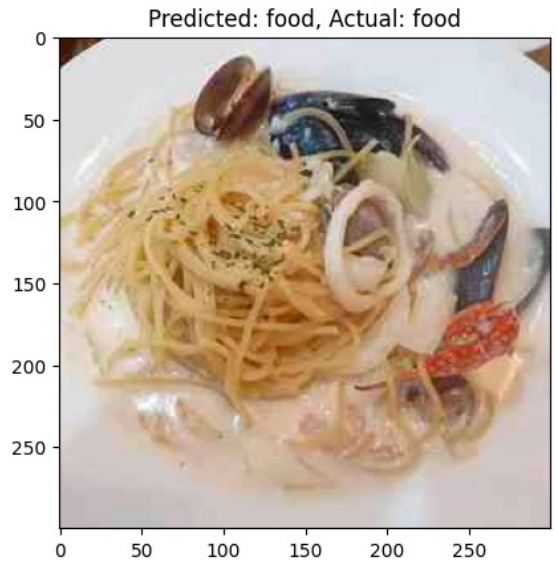
```
Epoch 1/2 - Train Loss: 0.9172, Validation Loss: 0.8718

Epoch 2/2 - Train Loss: 0.7801, Validation Loss: 0.7978
```

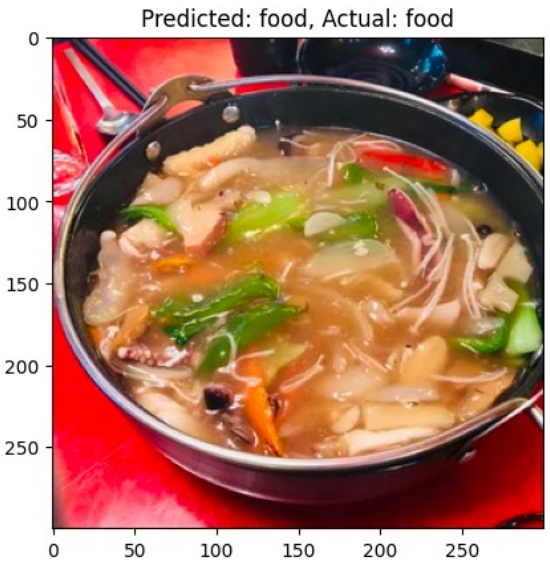


----Correct Predictions---- 10개

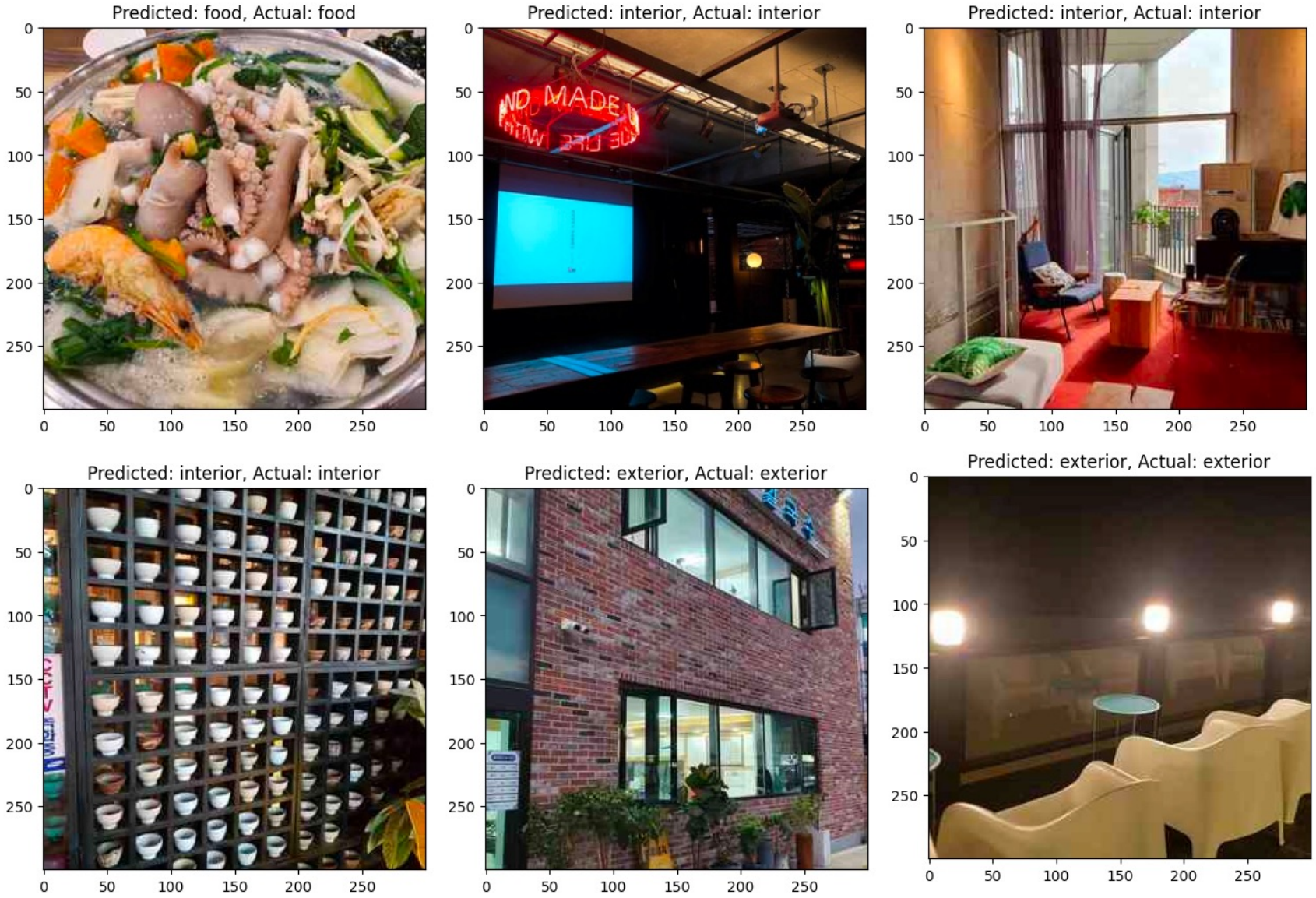




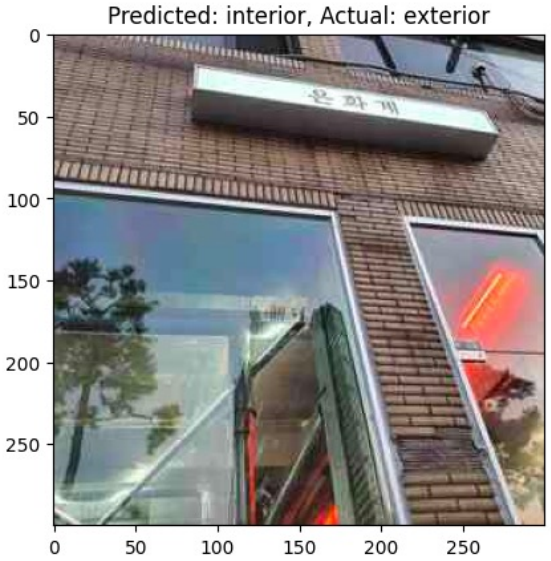


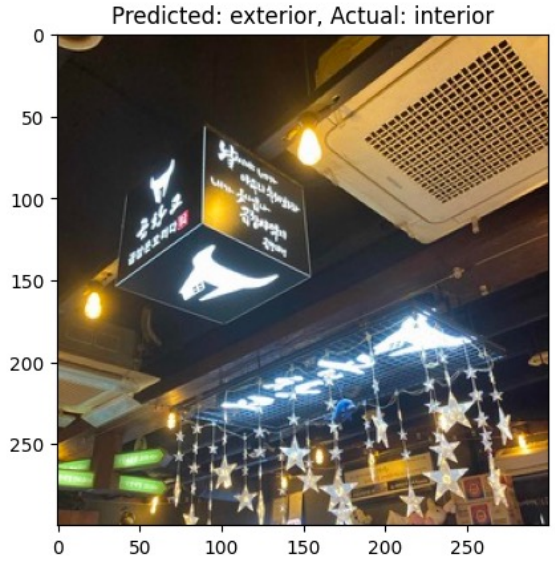


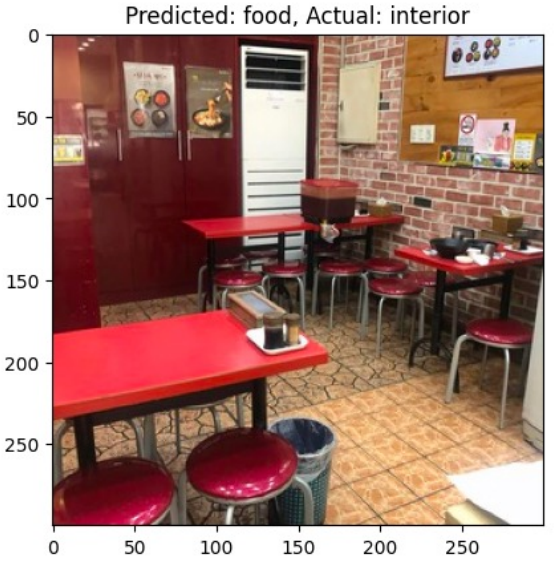
----Correct Predictions----



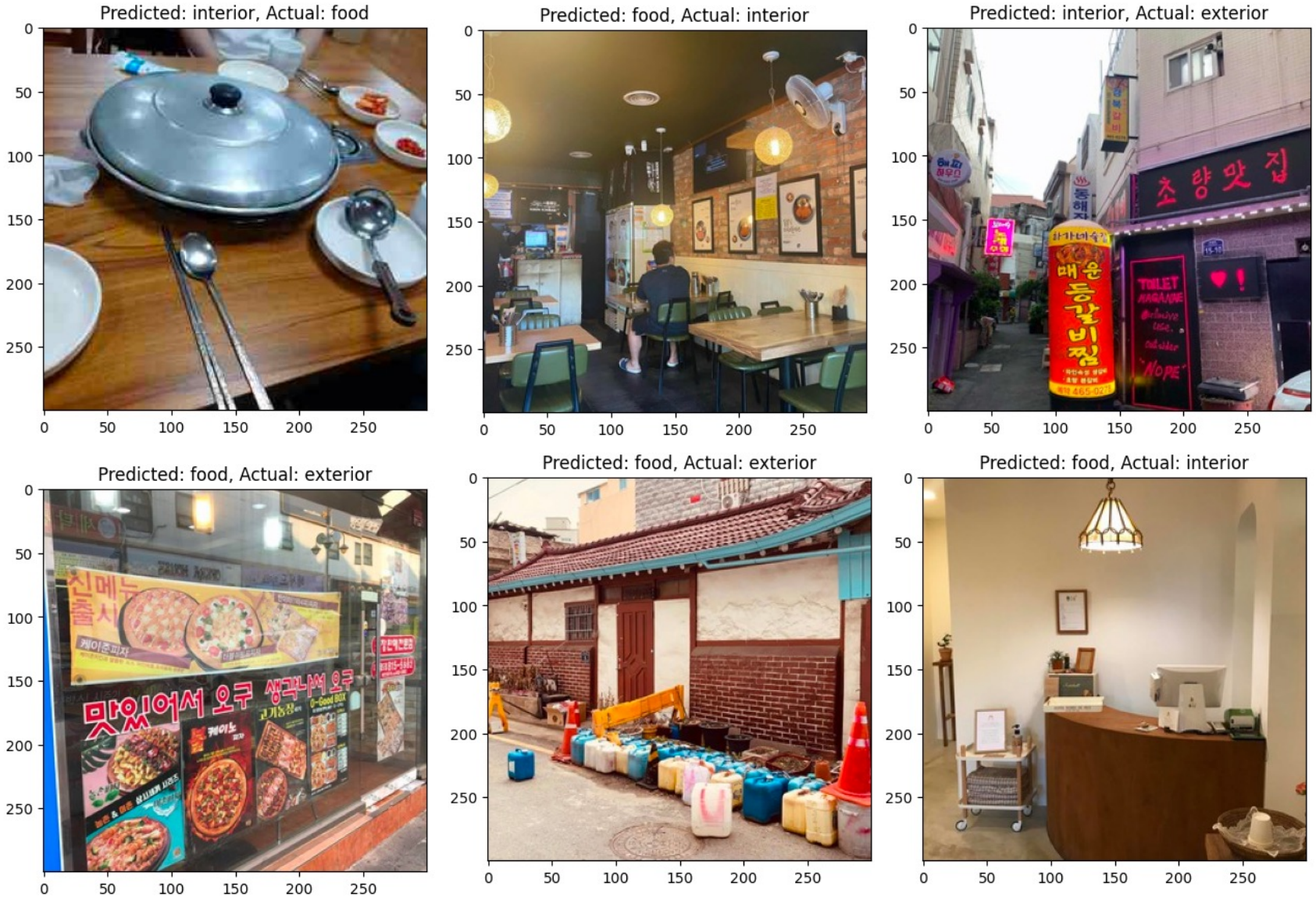
----Incorrect Predictions---- 10개







----Incorrect Predictions----



Accuracy & Confusion Matrix

Accuracy : 0.6367, 사용하기 많이 애매한 정확도

```
Accuracy =  0.6367
Test Loss: 0.8397
Confusion Matrix:
[[172  64  76]
 [ 42 513  88]
 [ 96 156 230]]
```

Convolution Neural Network

nn.Module

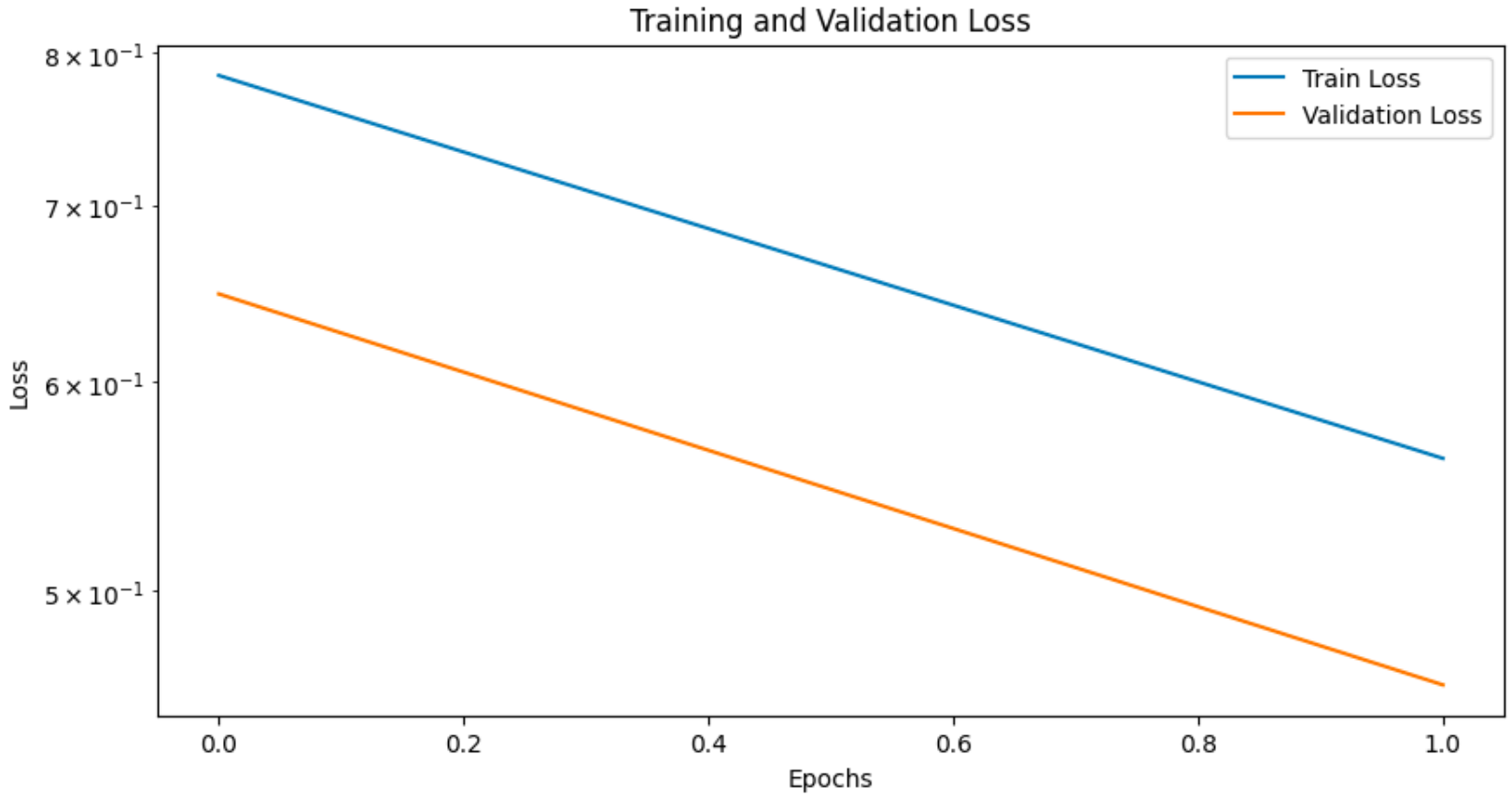
총 3개의 Convolution-Pooling 쌍, Stride=1 + Padding=1을 통해 kernel_size에 상응하는 input-output size matching

```python
class ImageClassificationModel_CNN(nn.Module):
    def __init__(self):
        super(ImageClassificationModel_CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
        self.conv3 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2, padding = 0)
        self.dropout = nn.Dropout(0.5)

        self.fc1 = nn.Linear(64 * 37 * 37, 64)
        self.fc2 = nn.Linear(64, 32)
        self.fc3 = nn.Linear(32, 3)

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = self.pool(torch.relu(self.conv3(x)))

        x = x.view(-1, 64 * 37 * 37)

        x = self.dropout(x)

        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

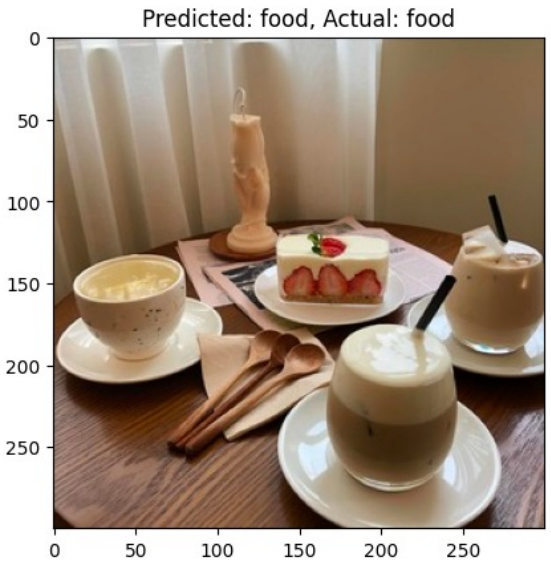
Dropout 이후 구성된
세 개의 FC

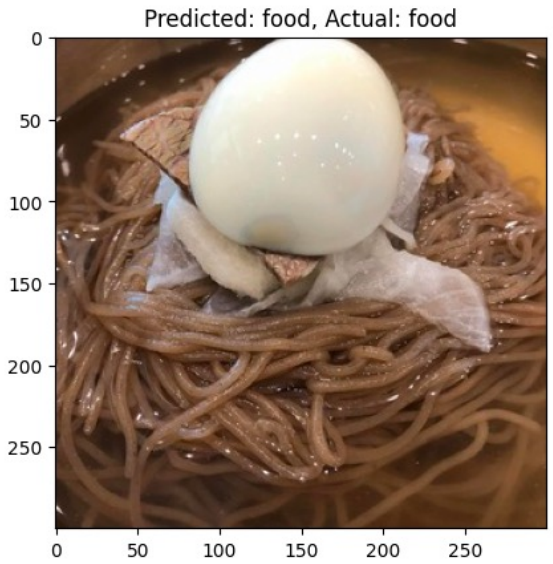
Plot Loss

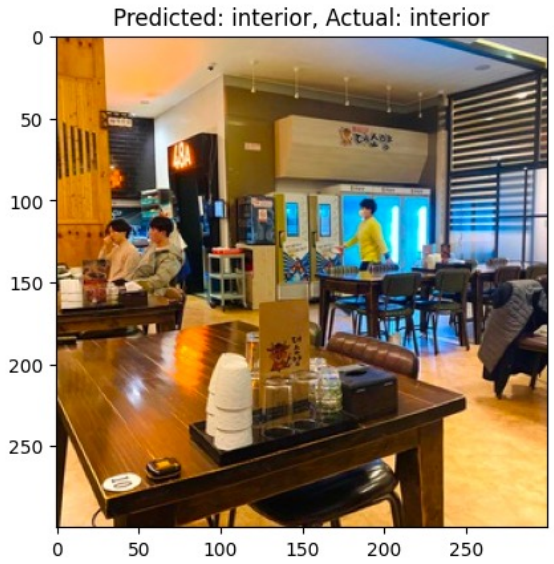
```
Epoch 1/2 - Train Loss: 0.7843, Validation Loss: 0.6481

Epoch 2/2 - Train Loss: 0.5615, Validation Loss: 0.4608
```

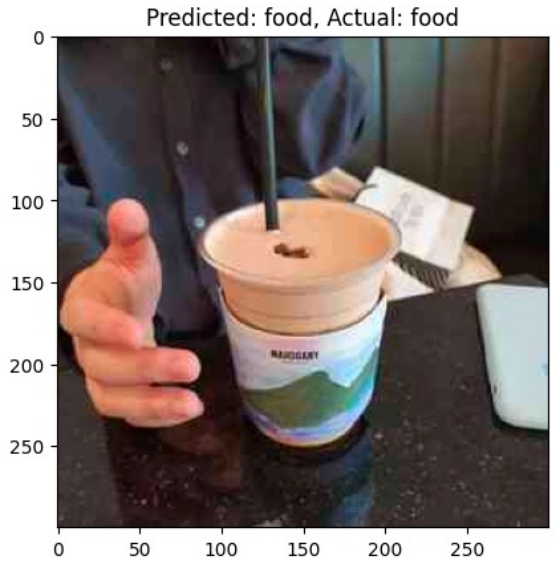


----Correct Predictions---- 10개

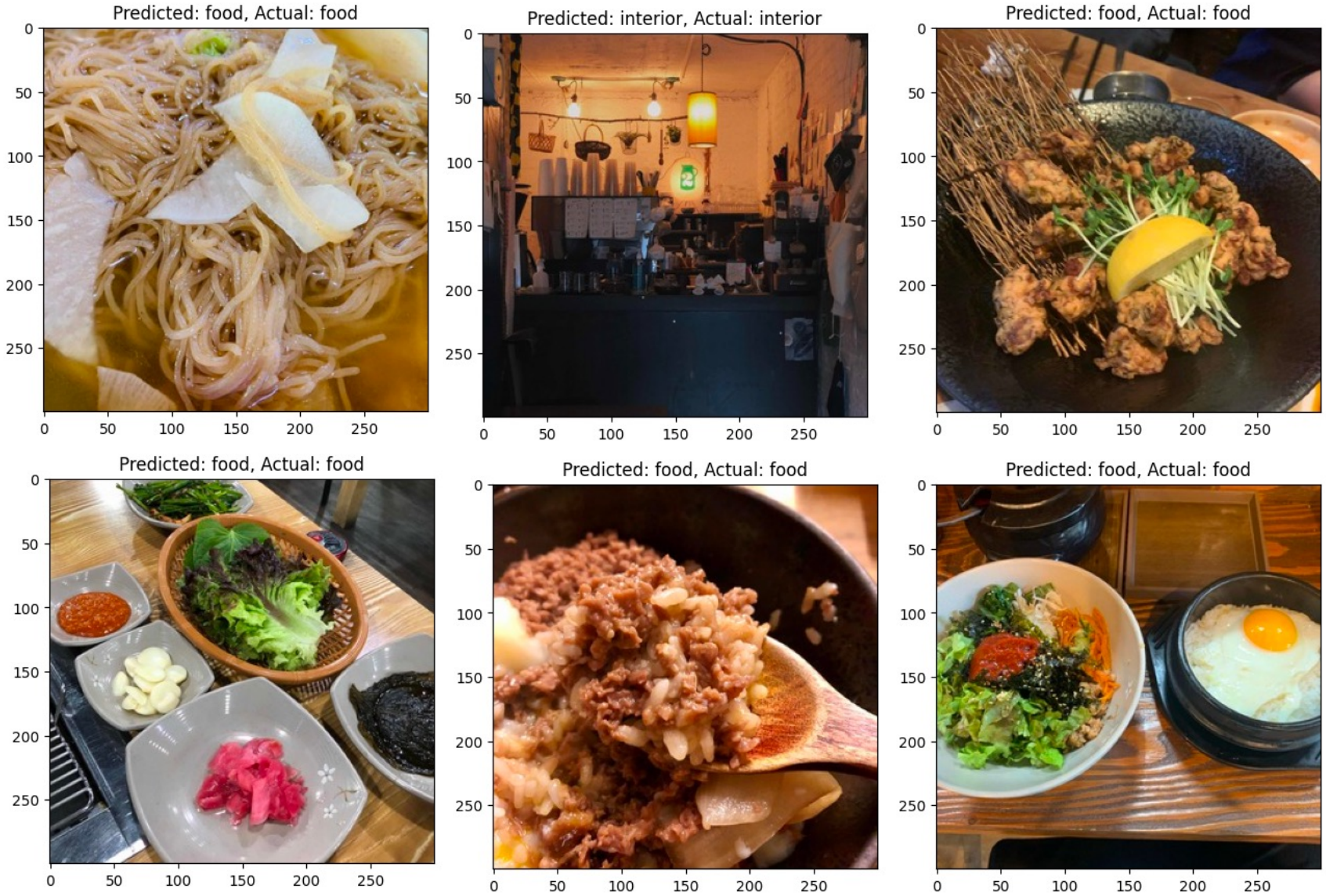




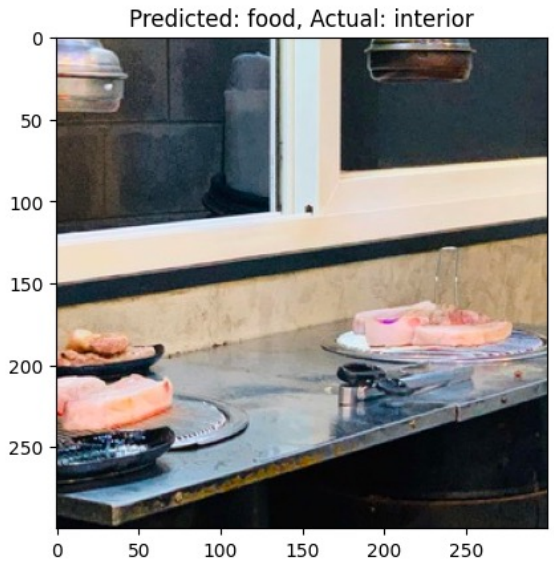


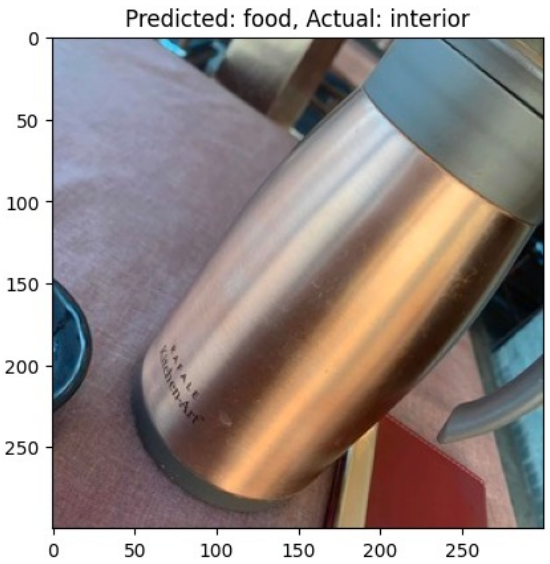


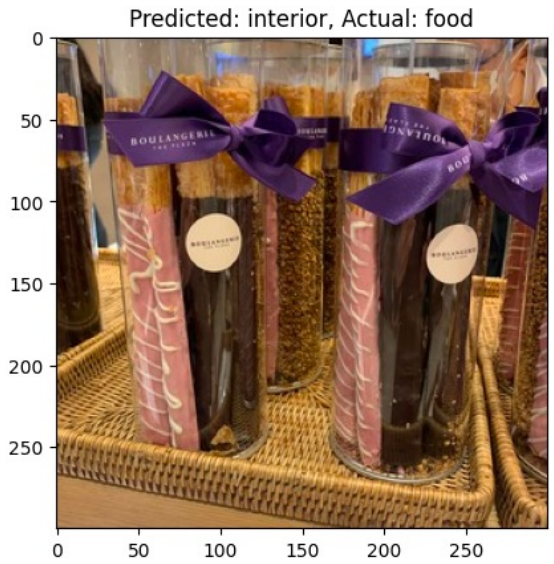
----Correct Predictions----

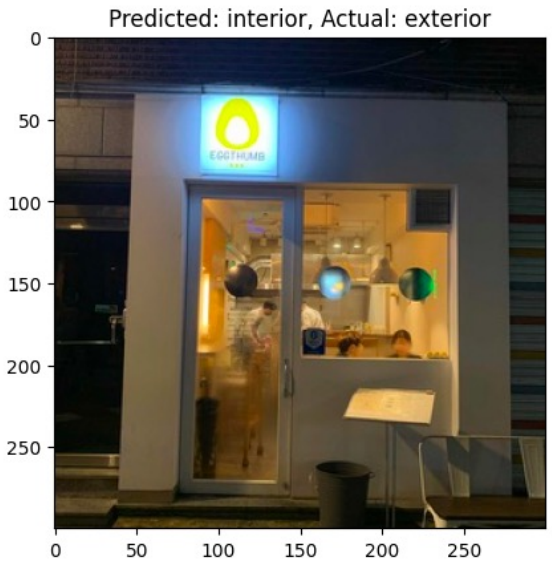


----Incorrect Predictions---- 10개

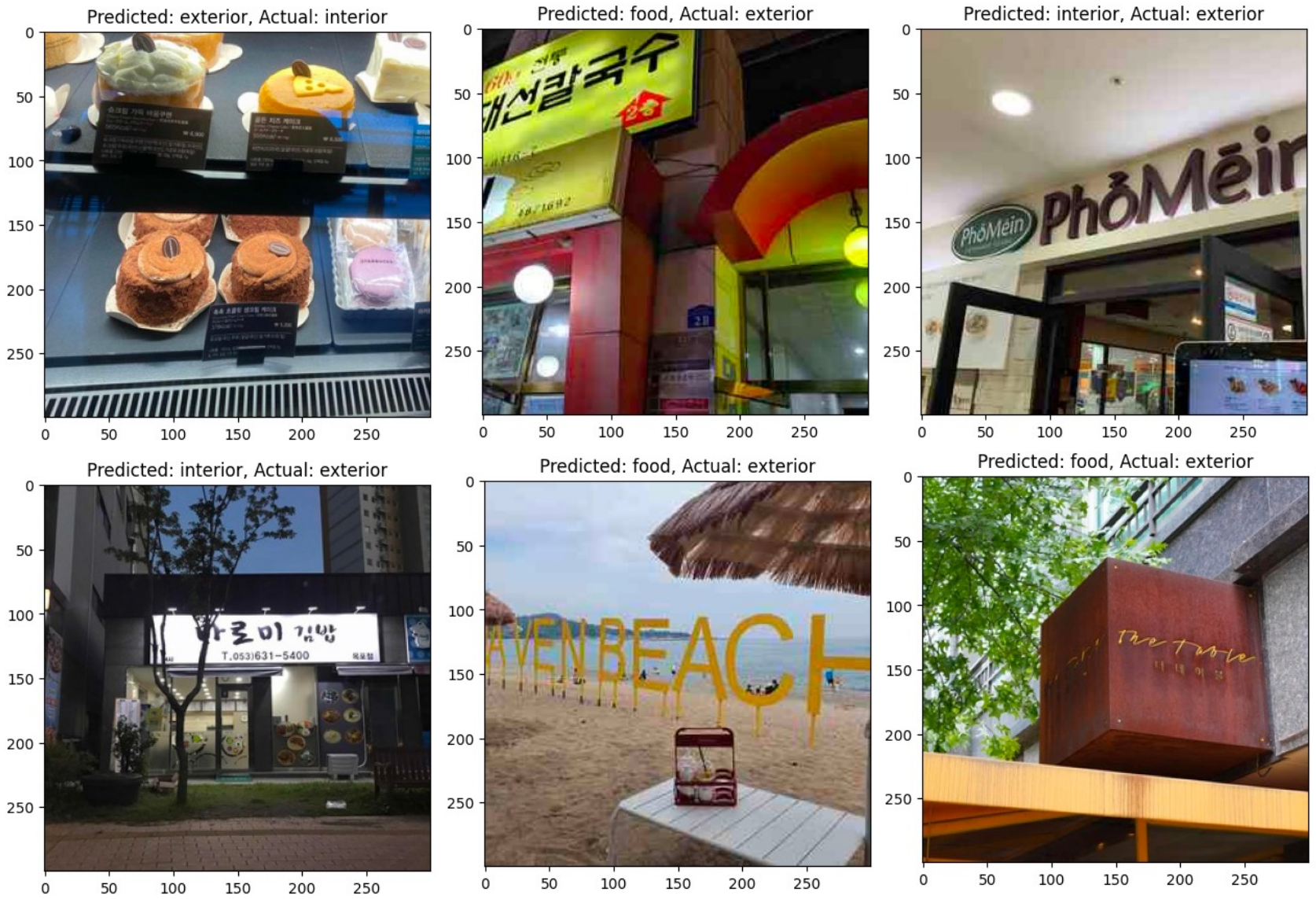








----Incorrect Predictions----



Accuracy & Confusion Matrix

Accuracy : 0.8107, FC_Only 때보다는 준수한 정확도, 여러 번의 cnn node 값 수정 trials의 결과

```
Accuracy =  0.8107
Test Loss: 0.4804
Confusion Matrix:
[[195  35  93]
 [  3 600  29]
 [ 32  80 370]]
```

Building Block Method

ResNet50(weights=ResNet50_Weights.DEFAULT)

Replace each 2-layer block in the 34-layer net with 3-layer bottleneck block, resulting in a 50-layer ResNet

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
...
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

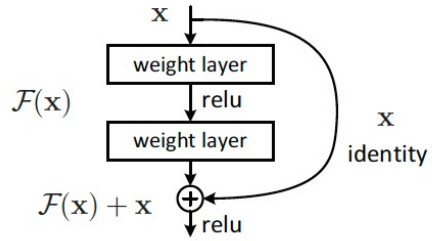


Figure 2. Residual learning: a building block.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
		3×3 max pool, stride 2				
conv2_x	56×56	[3×3, 64; 3×3, 64] ×2	[3×3, 64; 3×3, 64] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3	[1×1, 64; 3×3, 64; 1×1, 256] ×3
conv3_x	28×28	[3×3, 128; 3×3, 128] ×2	[3×3, 128; 3×3, 128] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×4	[1×1, 128; 3×3, 128; 1×1, 512] ×8
conv4_x	14×14	[3×3, 256; 3×3, 256] ×2	[3×3, 256; 3×3, 256] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×6	[1×1, 256; 3×3, 256; 1×1, 1024] ×23	[1×1, 256; 3×3, 256; 1×1, 1024] ×36
conv5_x	7×7	[3×3, 512; 3×3, 512] ×2	[3×3, 512; 3×3, 512] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3	[1×1, 512; 3×3, 512; 1×1, 2048] ×3
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

nn.Module

Pre-Trained model의 features를 identity matrix로 받고, FC layer로 연결, fc1은 requires_grad -> FALSE

이후 fc2, fc3는 모두 TRUE로 받고 training을 진행, 여러 번의 시행 결과 dropout을 제외했을 때 좋은 성능을 보임.

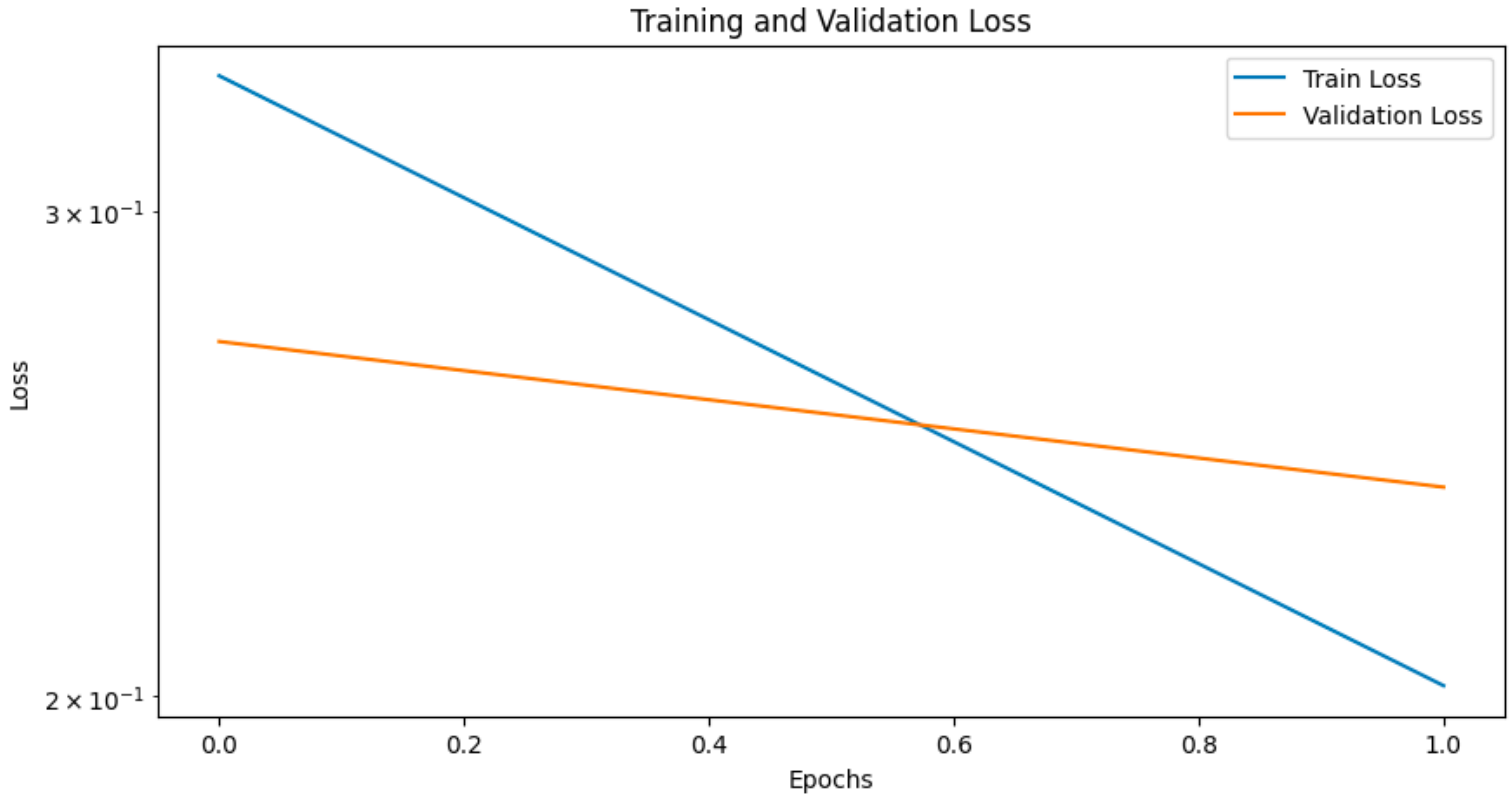
```python
class building_block_method(nn.Module):
    def __init__(self, num_classes, fc_requires_grad=True):
        super(building_block_method, self).__init__()

        resnet_model = resnet50(weights=ResNet50_Weights.DEFAULT)
        in_features = resnet_model.fc.in_features
        resnet_model.fc = nn.Identity()

        for param in resnet_model.parameters():
            param.requires_grad = False

        self.resnet = resnet_model
        self.fc1 = nn.Linear(in_features, 32)
        self.fc1.requires_grad = False
        self.fc2 = nn.Linear(32, 64)
        self.fc2.requires_grad = fc_requires_grad
        self.fc3 = nn.Linear(64, num_classes)
        self.fc3.requires_grad = fc_requires_grad

    def forward(self,x):
        x = self.resnet(x)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)

        return x
```

Pre-Trained

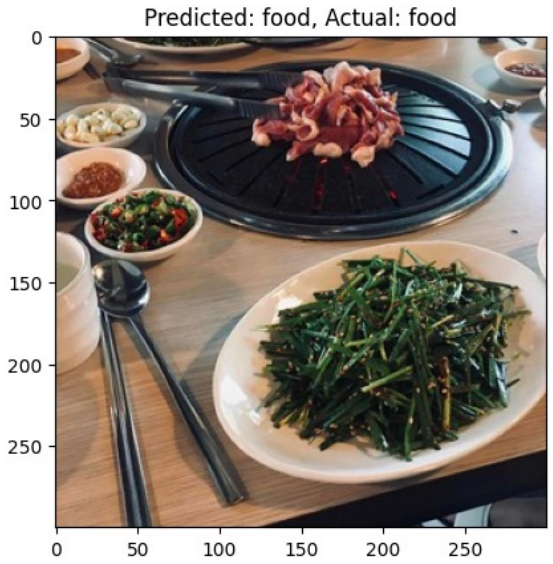
Plot Loss

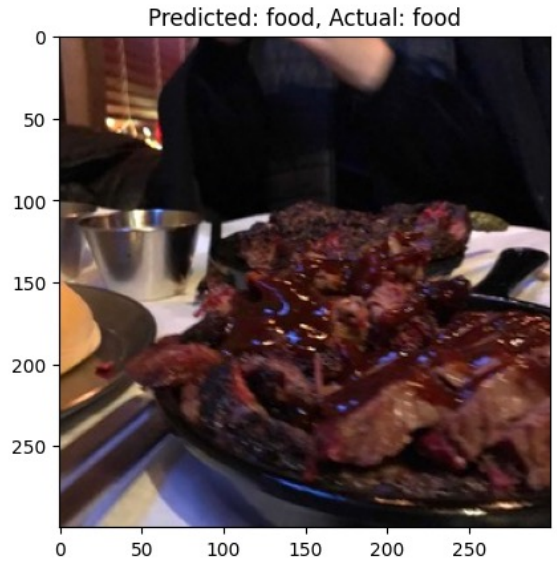
전체적으로 준수하게 감소된 Loss 수준을 보임.

```
Epoch 1/2 - Train Loss: 0.3357, Validation Loss: 0.2690

Epoch 2/2 - Train Loss: 0.2018, Validation Loss: 0.2382
```

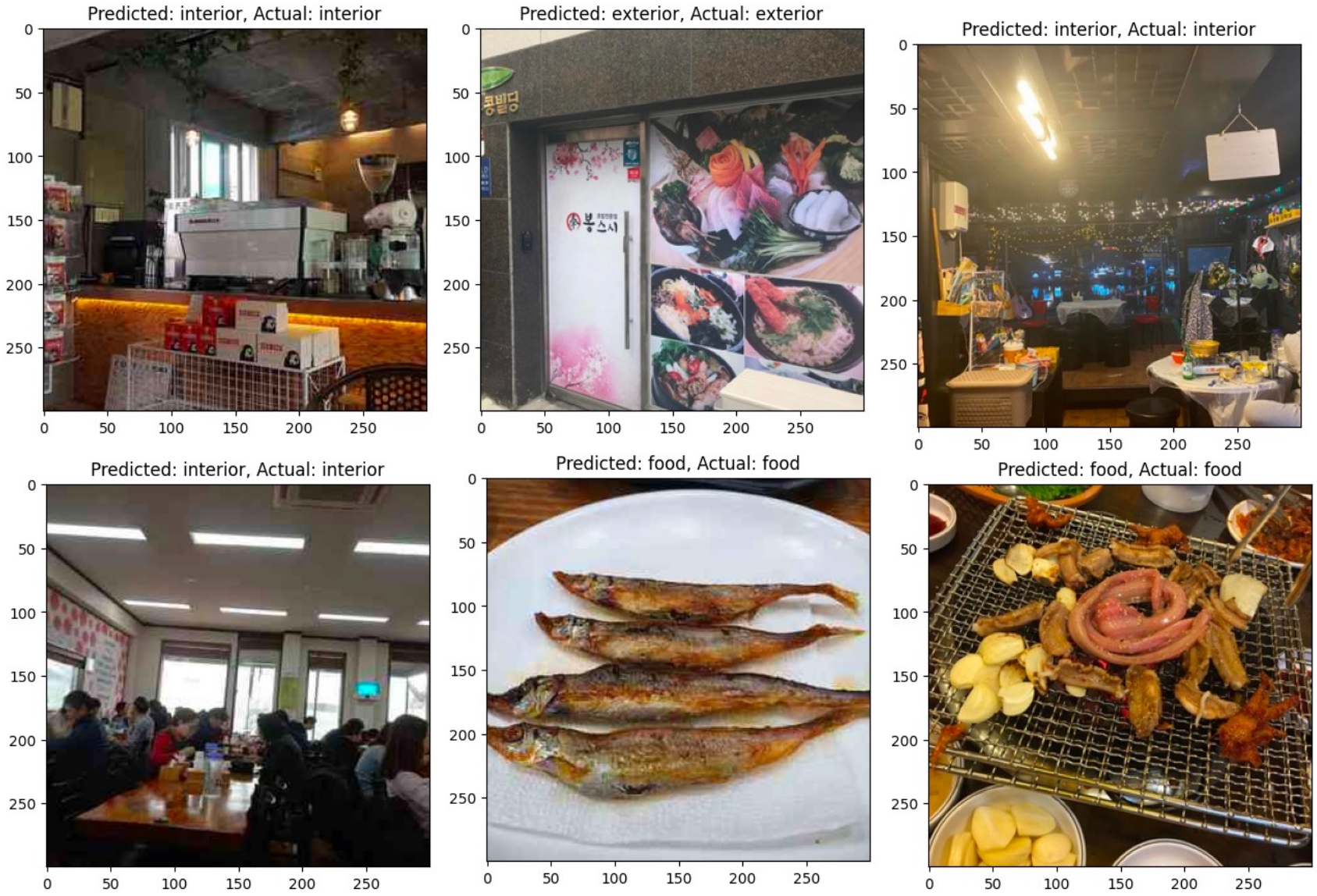


----Correct Predictions---- 10개

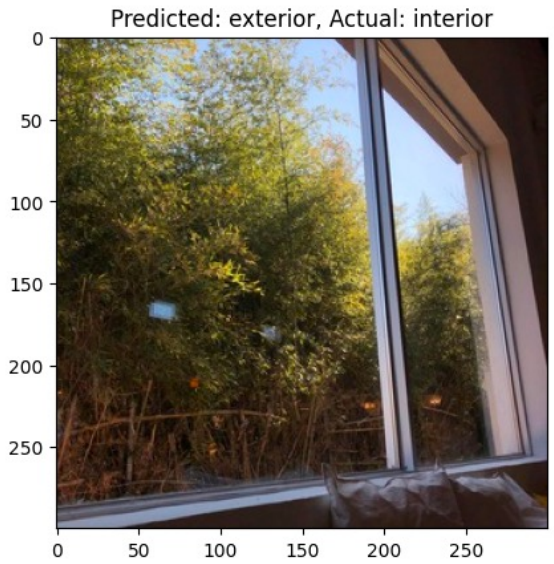


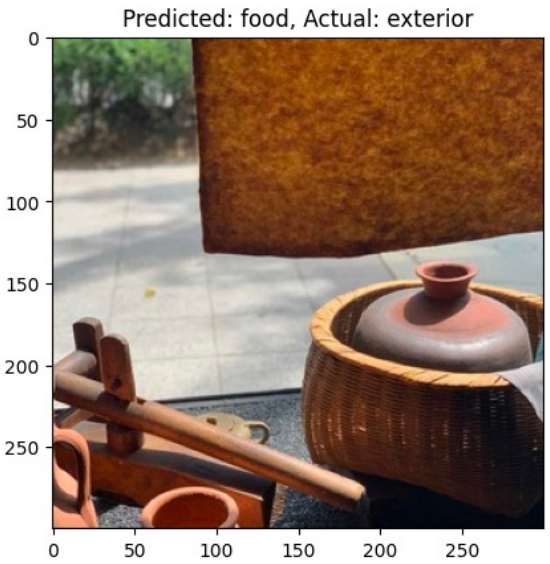


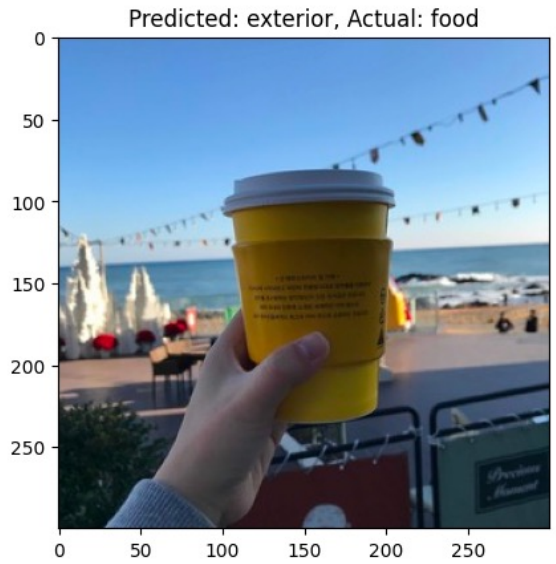
----Correct Predictions----

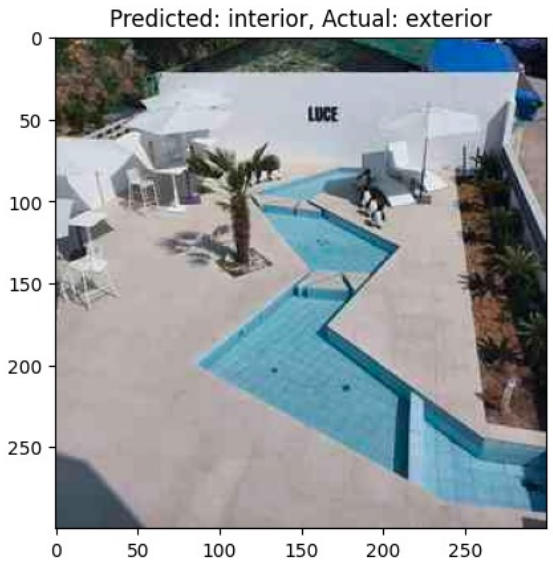


----Incorrect Predictions---- 10개









----Incorrect Predictions----

Accuracy & Confusion Matrix

Accuracy : 0.9374, 세 개의 method 중 가장 준수한 정확도,

추가적으로, ResNet50_Weights.DEFAULT을 활용한 building block method는
resnet50 구조의 **전체 parameter**를 학습 했을 때의 accuracy와 거의 유사한 값을 산출했음.
따라서, Generalization이 매우 잘 되어있는 Parameter임을 확인할 수 있음.

```
Accuracy =  0.9374
Test Loss: 0.1725
Confusion Matrix:
[[269   2  33]
 [  1 629   6]
 [ 18  30 449]]
```

End