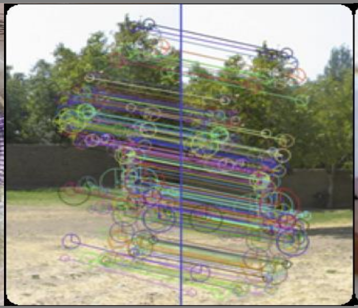
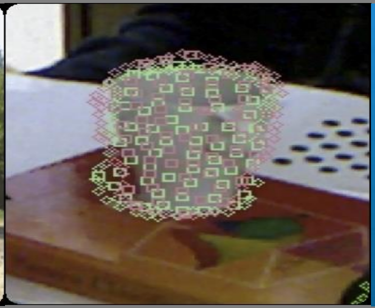
Visual Intelligence Theory
(#13: Keras-based Convolutional Neural Network Practice-Part 8)
Transfer Learning#2 – Refinement (Fine-tuning)

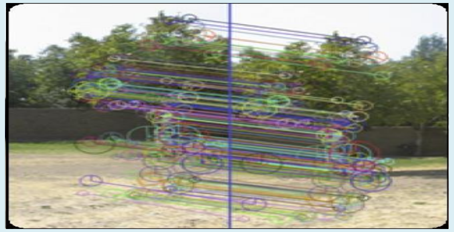




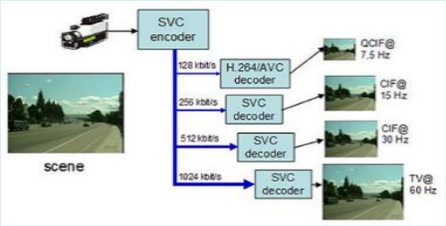
2023 Fall

Prof. Byung-Gyu Kim
Intelligent Vision Processing Lab. (IVPL)
http://ivpl.sookmyung.ac.kr
Dept. of IT Engineering, Sookmyung Women's University
E-mail: bg.kim@ivpl.sookmyung.ac.kr

Goal of this lecture

- Understanding what is the transfer learning
  - Transfer learning as refinement (Fine-tuning)
  - How to implement the transfer learning using refinement (Fine-tuning)
  - Actual practice





Contents

- Transfer Learning-Refinement (Fine-tuning)

Transfer Learning (1)

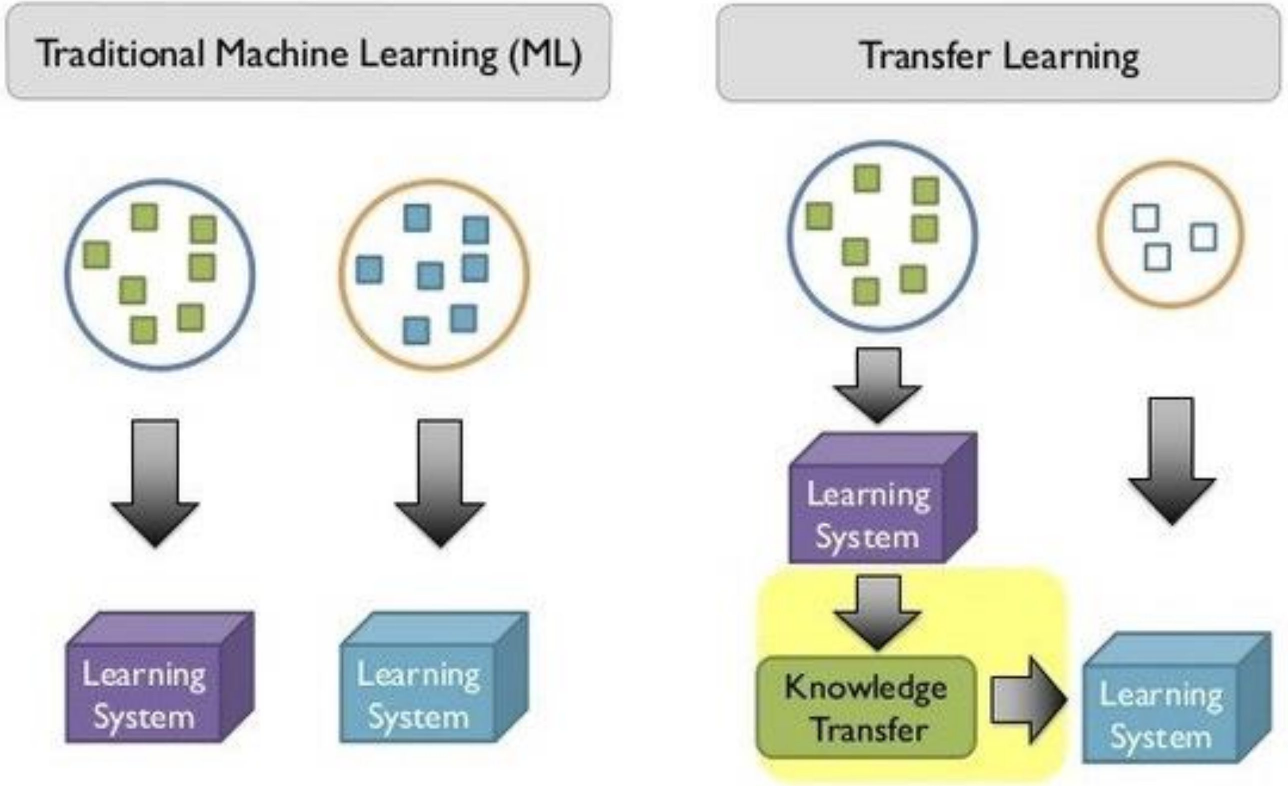
❖ What is "**Transfer Learning**"?

- When a new object recognition or classification is required using the previously learned (trained) object identification model.

EX) How to create an automated computer vision application that can distinguish between "kinds of foods". Which way is the best????

Transfer Learning (2)

- Two ways:
  1) **New model generation** (New training)
  2) **Utilize the pre-trained model** to get some results



Transfer Learning (3) – Fine-tuning : using Keras

❖ **Transfer Learning** is composed of:

1) Taking a network *pre-trained* on a dataset.
   - Utilize the robust, discriminative filters learned by state-of-the-art networks on challenging datasets (such as ImageNet or COCO).
2) And utilizing it to recognize image/object categories it was not trained on.
   - then apply these networks to recognize objects the model was *never trained* on.

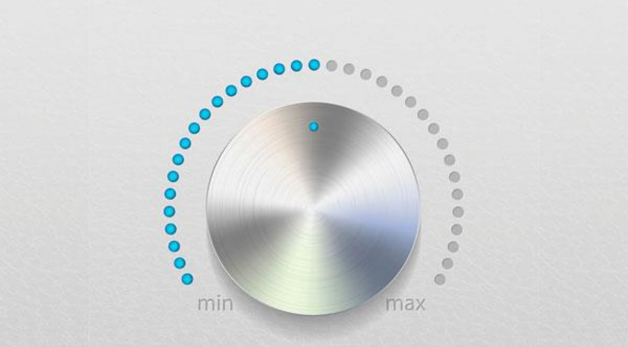
Transfer Learning (4) – Fine-tuning : using Keras

❖ **Two types of transfer learning** in the context of deep learning:

1) Transfer learning via **feature extraction**
2) Transfer learning via **fine-tuning**

In *feature extraction*, we treat the pre-trained network as an arbitrary feature extractor, **allowing the input image to propagate forward, stopping at pre-specified layer, and taking the *outputs* of that layer** as your features.

Fine-tuning, on the other hand, requires that we **update the model architecture itself by removing the previous fully-connected layer heads, providing new, freshly initialized ones, and then training the new FC layers to predict our input classes.**



❖ Refinement (Fine-tuning) Approach

- **Fine-tuning requires that we not only *update* the CNN architecture but also *re-train* it to learn new object classes.**

❖ Fine-tuning Process:

1) **Remove the fully connected nodes** at the end of the network (i.e., where the actual class label predictions are made).
2) **Replace the fully connected nodes with freshly initialized ones**.
3) **Freeze earlier CONV layers earlier in the network** (ensuring that any previous robust features learned by the CNN are not destroyed).
4) Start training, *but **only train the FC layer heads***.
5) Optionally **unfreeze some/all of the CONV layers in the network and perform a second pass of training.**

Transfer Learning (5) – Fine-tuning : using Keras

❖ Refinement (Fine-tuning) Approach

- 1) Datasets
  - **The dataset consists of 16,643 images belonging to 11 major food categories**: (https://mmspg.epfl.ch/downloads/food-image-datasets/) curated by the Multimedia Signal Processing Group (MSPG) of the Swiss Federal Institute of Technology.
    (You can use FTP client program to download Food-11 dataset.)

1. **Bread** (1724 images)
2. **Dairy product** (721 images)
3. **Dessert** (2,500 images)
4. **Egg** (1,648 images)
5. **Fried food** (1,461images)
6. **Meat** (2,206 images)
7. **Noodles/pasta** (734 images)
8. **Rice** (472 images)
9. **Seafood** (1,505 images)
10. **Soup** (2,500 images)
11. **Vegetable/fruit** (1,172 images)

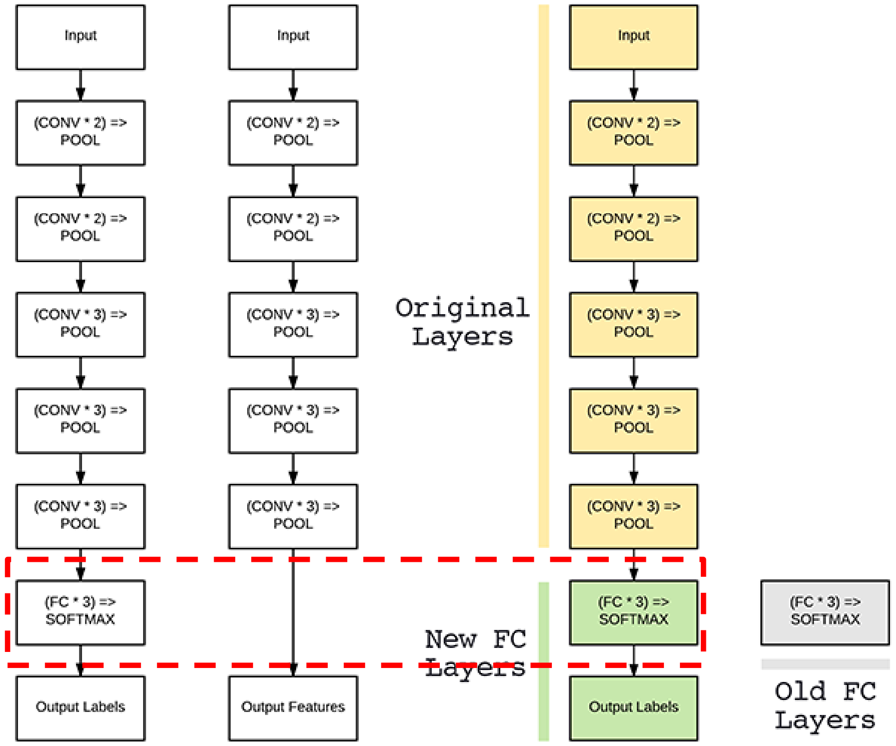
[the Food-11 dataset]

Transfer Learning (6) – Fine-tuning : using Keras

- 2) Train the CNN, first..!!!
  - Deep neural networks trained on large-scale datasets such as **ImageNet** and **COCO** have proven to be *excellent* at the task of transfer learning.
  - These networks learn a set of rich, discriminative features capable of recognizing 100s to 1,000s of object classes — it only makes sense that these filters can be reused for tasks other than what the CNN was originally trained on (VGG, ResNet, or Inception).

The original VGG16 network architecture

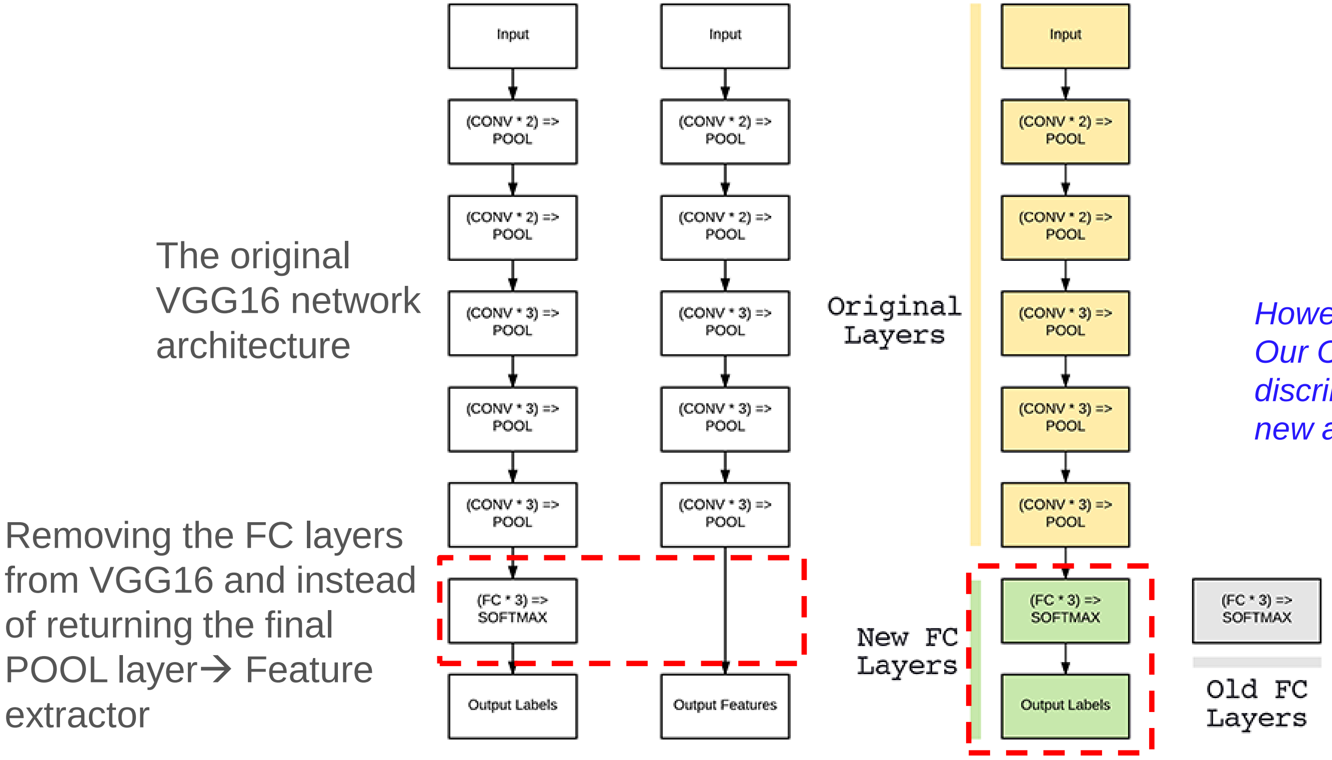
Removing the FC layers from VGG16 and instead of returning the final POOL layer→ Feature extractor



[Feature extractor] [Refinement (Fine-tuning)]

Transfer Learning (7) – Fine-tuning : using Keras

- 3) We are going to perform network surgery and ***modify the actual architecture*** so that we can re-train parts of the network.
  - Remove the original fully connected (FC) networks.
  - **Build a new fully connected (FC) networks and place it on top of the original architecture (right of the below figure).**

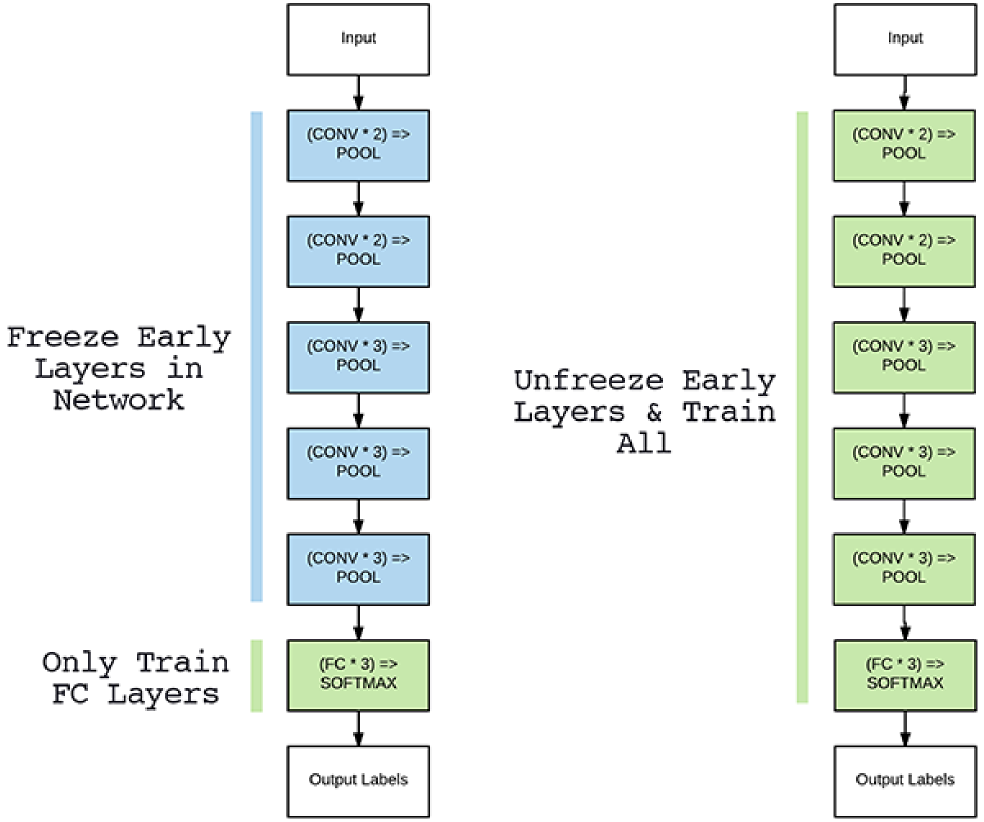


[Feature extractor] [Refinement (Fine-tuning)]

However, there is a problem:
Our CONV layers have already learned rich, discriminative filters while our FC layers are brand new and totally random.

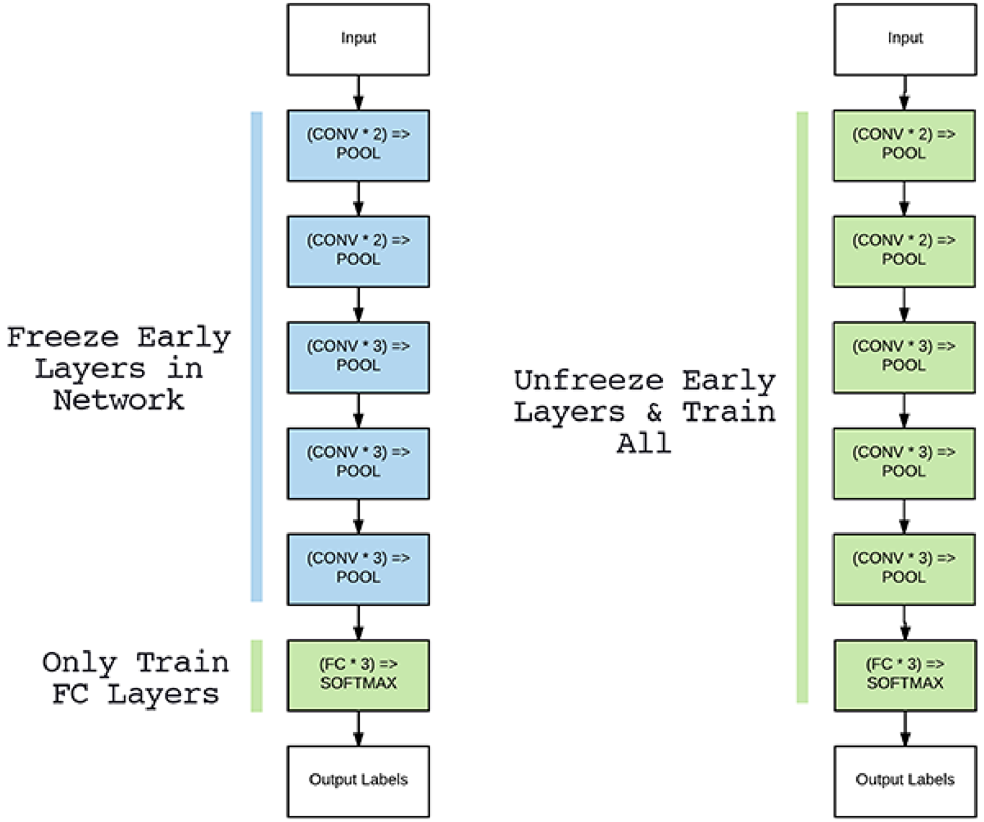
Transfer Learning (8) – Fine-tuning : using Keras

- 4) By (ironically) “**freezing**” all layers in the body of the network as depicted in **Figure** (*left*).
  - **Freezing** Layers: retain the feature weights of convolution networks
    - *Training data is forward propagated through the network as we usually would; however, the backpropagation is stopped after the FC layers*, which allows these layers to start to learn patterns from the highly discriminative CONV layers.



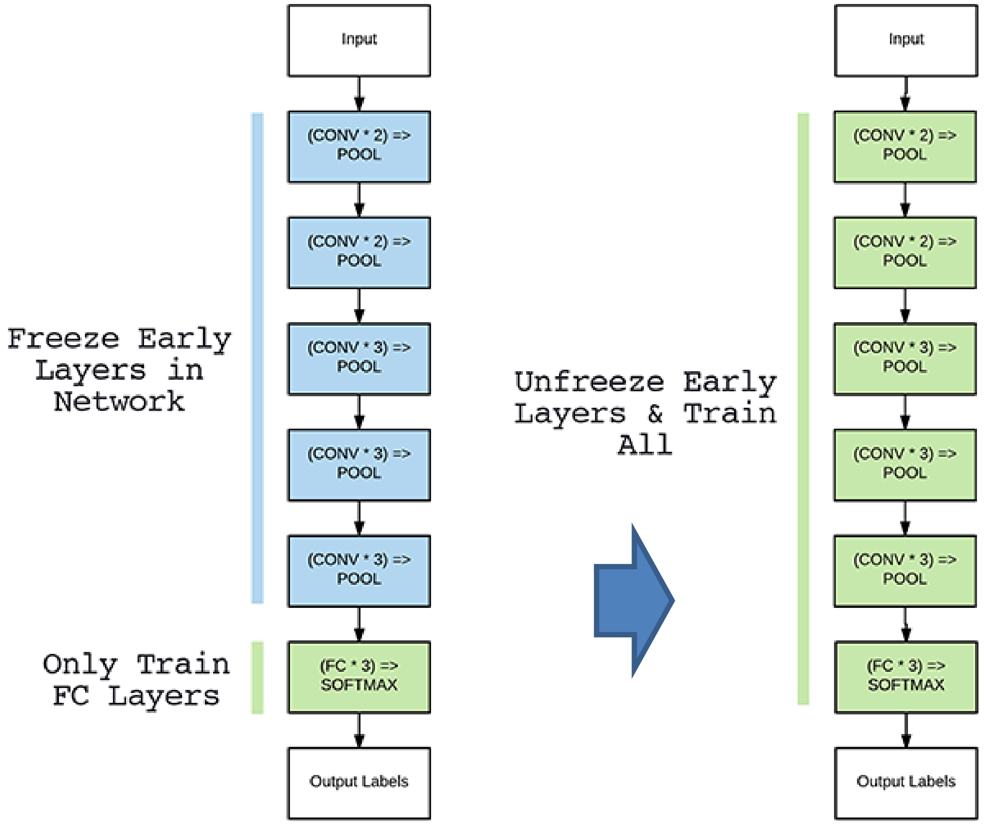
Transfer Learning (9) – Fine-tuning : using Keras

- 4) By (ironically) "**freezing**" all layers in the body of the network as depicted in **Figure** (*left*).
  - In some cases, we may decide to never unfreeze the body of the network as our new FC head may obtain sufficient accuracy.
  - However, for some datasets it is often advantageous to allow the original CONV layers to be modified during the fine-tuning process as well (Figure, *right*).



Transfer Learning (10) – Fine-tuning : using Keras

- 5) **After the FC head has started to learn patterns in our dataset**, **we can pause training, unfreeze the body, and continue training, *but with a very small learning rate*** — we do not want to alter our CONV filters dramatically.
  - Training is then allowed to continue until sufficient accuracy is obtained.



❖ Project structure

```
$ tree --dirsfirst --filelimit 10
.
├── Food-11
│   ├── evaluation [3347 entries]
│   ├── training [9866 entries]
│   ├── validation [3430 entries]
│   └── Food-11.zip
├── dataset
├── output
│   ├── unfrozen.png
│   └── warmup.png
├── pyimagesearch
│   ├── __init__.py
│   └── config.py
├── build_dataset.py
├── predict.py
└── train.py

7 directories, 8 files
```

dataset/ directory, while empty now, will soon contain the Food-11 images in a more organized form.
output/ directory will house our extracted features (stored in three separate .csv files).

- **pyimagesearch/config.py** : Our custom configuration file will help us manage our dataset, class names, and paths. It is written in Python directly so that we can use os.path to build OS-specific formatted file paths directly in the script.
- **build_dataset.py** : Using the configuration, this script will create an organized dataset on disk, making it easy to extract features from to dataset directory.
- predict.py : to make predictions on sample images using our fine-tuned network.
- **train.py** : Our training script will perform fine-tuning.

Transfer Learning (12) – Fine-tuning : Actual Practice – Foods classification (2)

- config.py

```
# import the necessary packages
import os

# initialize the path to the *original* input directory of
images
ORIG_INPUT_DATASET = "Food-11"

# initialize the base path to the *new* directory that will
contain
# our images after computing the training and testing split
BASE_PATH = "dataset"

# define the names of the training, testing, and validation
# directories
TRAIN = "training"
TEST = "evaluation"
VAL = "validation"

# initialize the list of class label names
CLASSES = ["Bread", "Dairy product", "Dessert", "Egg",
"Fried food",
"Meat", "Noodles/Pasta", "Rice", "Seafood", "Soup",
"Vegetable/Fruit"]

# set the batch size when fine-tuning
BATCH_SIZE = 32
```

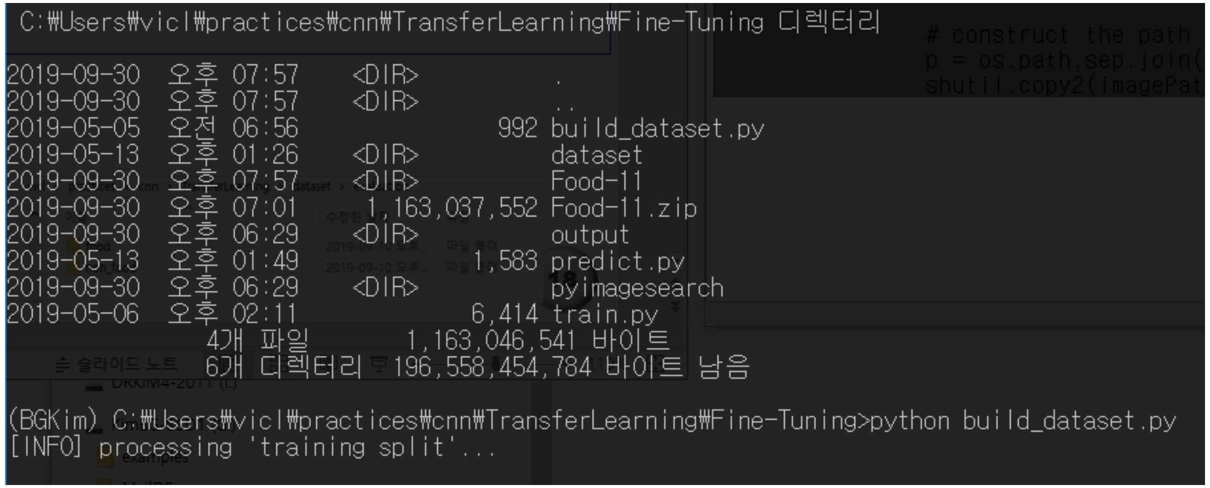
```
      (continue)
# set the batch size when fine-tuning
BATCH_SIZE = 32

# initialize the label encoder file path and the output directory
to
# where the extracted features (in CSV file format) will be stored
LE_PATH = os.path.sep.join(["output", "le.cpickle"])
BASE_CSV_PATH = "output"

# set the path to the serialized model after training
MODEL_PATH = os.path.sep.join(["output", "food11.model"])

# define the path to the output training history plots
UNFROZEN_PLOT_PATH = os.path.sep.join(["output", "unfrozen.png"])
WARMUP_PLOT_PATH = os.path.sep.join(["output", "warmup.png"])
```

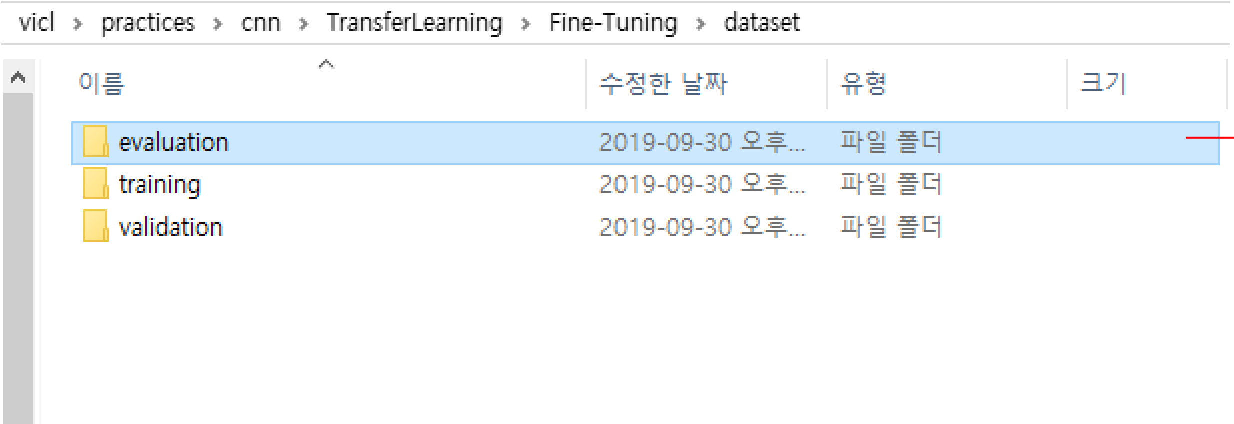
IVPL
Intelligent Vision Processing Lab

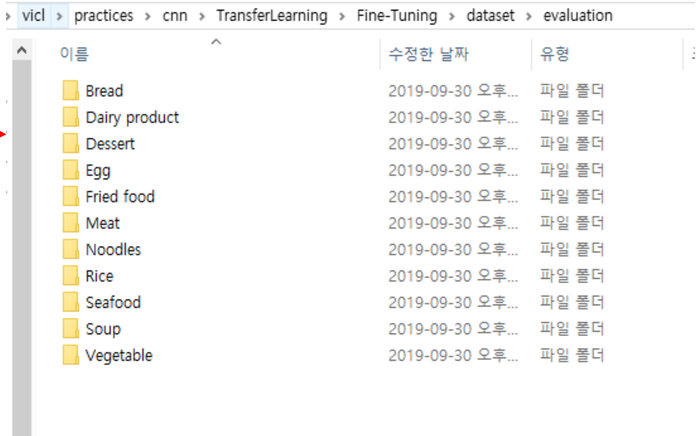
- build_dataset.py

```
# import the necessary packages
from pyimagesearch import config
from imutils import paths
import shutil
import os

# loop over the data splits
for split in (config.TRAIN, config.TEST, config.VAL):
    # grab all image paths in the current split
    print("[INFO] processing '{} split'...".format(split))
    p = os.path.sep.join([config.ORIG_INPUT_DATASET, split])
    imagePaths = list(paths.list_images(p))

    # loop over the image paths
    for imagePath in imagePaths:
        # extract class label from the filename
        filename = imagePath.split(os.path.sep)[-1]
        label = config.CLASSES[int(filename.split("_")[0])]

        # construct the path to the output directory
        dirPath = os.path.sep.join([config.BASE_PATH, split, label])

        # if the output directory does not exist, create it
        if not os.path.exists(dirPath):
        os.makedirs(dirPath)

        # construct the path to the output image file and copy it
        p = os.path.sep.join([dirPath, filename])
        shutil.copy2(imagePath, p)
```

- build_dataset.py







- train.py(1)

```
# set the matplotlib backend so figures can be saved in the
background
import matplotlib
matplotlib.use("Agg")

# import the necessary packages
from keras.preprocessing.image import ImageDataGenerator
from keras.applications import VGG16
from keras.layers.core import Dropout
from keras.layers.core import Flatten
from keras.layers.core import Dense
from keras.layers import Input
from keras.models import Model
from keras.optimizers import SGD
from sklearn.metrics import classification_report
from pyimagesearch import config
from imutils import paths
import matplotlib.pyplot as plt
import numpy as np
import pickle
import os
```

(continue)

```
def plot_training(H, N, plotPath):
# construct a plot that plots and saves the training history
    plt.style.use("ggplot")
    plt.figure()
    plt.plot(np.arange(0, N), H.history["loss"],
     label="train_loss")
    plt.plot(np.arange(0, N), H.history["val_loss"],
     label="val_loss")
    plt.plot(np.arange(0, N), H.history["acc"],
     label="train_acc")
    plt.plot(np.arange(0, N), H.history["val_acc"],
     label="val_acc")
    plt.title("Training Loss and Accuracy")
    plt.xlabel("Epoch #")
    plt.ylabel("Loss/Accuracy")
    plt.legend(loc="lower left")
    plt.savefig(plotPath)
```

- train.py (2)

```
# derive the paths to the training, validation, and
testing
# directories
trainPath = os.path.sep.join([config.BASE_PATH,
config.TRAIN])
valPath = os.path.sep.join([config.BASE_PATH, config.VAL])
testPath = os.path.sep.join([config.BASE_PATH,
config.TEST])

# determine the total number of image paths in training,
validation,
# and testing directories
totalTrain = len(list(paths.list_images(trainPath)))
totalVal = len(list(paths.list_images(valPath)))
totalTest = len(list(paths.list_images(testPath)))

# initialize the training data augmentation object
trainAug = ImageDataGenerator(
               rotation_range=30,
               zoom_range=0.15,
               width_shift_range=0.2,
               height_shift_range=0.2,
               shear_range=0.15,
               horizontal_flip=True,
               fill_mode="nearest")
```

```
# initialize the validation/testing data augmentation object
#(which we'll be adding mean subtraction to)
valAug = ImageDataGenerator()

# define the ImageNet mean subtraction (in RGB order) and set
the
# the mean subtraction value for each of the data
augmentation
# objects
mean = np.array([123.68, 116.779, 103.939], dtype="float32")
trainAug.mean = mean
valAug.mean = mean

# initialize the training generator
trainGen = trainAug.flow_from_directory(
          trainPath,
          class_mode="categorical",
          target_size=(224, 224),
          color_mode="rgb",
          shuffle=True,
          batch_size=config.BATCH_SIZE)
```

- train.py (3)

```python
# initialize the validation generator
valGen = valAug.flow_from_directory(
	valPath,
	class_mode="categorical",
	target_size=(224, 224),
	color_mode="rgb",
	shuffle=False,
	batch_size=config.BATCH_SIZE)

# initialize the testing generator
testGen = valAug.flow_from_directory(
		testPath,
		class_mode="categorical",
		target_size=(224, 224),
		color_mode="rgb",
		shuffle=False,
		batch_size=config.BATCH_SIZE)

		(continue)
```

```python
# load the VGG16 network, ensuring the head FC layer sets are left
# off
baseModel = VGG16(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(512, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(len(config.CLASSES), activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
		layer.trainable = False
```

IVPL
Intelligent Vision Processing Lab

- train.py (4)

```
# initialize the validation generator
valGen = valAug.flow_from_directory(
	valPath,
	class_mode="categorical",
	target_size=(224, 224),
	color_mode="rgb",
	shuffle=False,
	batch_size=config.BATCH_SIZE)

# initialize the testing generator
testGen = valAug.flow_from_directory(
		testPath,
		class_mode="categorical",
		target_size=(224, 224),
		color_mode="rgb",
		shuffle=False,
		batch_size=config.BATCH_SIZE)

		(continue)
```

```
# load the VGG16 network, ensuring the head FC layer sets are left
# off
baseModel = VGG16(weights="imagenet", include_top=False,
input_tensor=Input(shape=(224, 224, 3)))

# construct the head of the model that will be placed on top of the
# the base model
headModel = baseModel.output
headModel = Flatten(name="flatten")(headModel)
headModel = Dense(512, activation="relu")(headModel)
headModel = Dropout(0.5)(headModel)
headModel = Dense(len(config.CLASSES), activation="softmax")(headModel)

# place the head FC model on top of the base model (this will become
# the actual model we will train)
model = Model(inputs=baseModel.input, outputs=headModel)

# loop over all layers in the base model and freeze them so they will
# *not* be updated during the first training process
for layer in baseModel.layers:
	layer.trainable = False
```

Intelligent Vision Processing Lab

- train.py (5)

```
# compile our model (this needs to be done after our setting our
# layers to being non-trainable
print("[INFO] compiling model...")
opt = SGD(lr=1e-4, momentum=0.9)
model.compile(loss="categorical_crossentropy", optimizer=opt,
        metrics=["accuracy"])

# train the head of the network for a few epochs (all other layers
# are frozen) -- this will allow the new FC layers to start to become
# initialized with actual "learned" values versus pure random
print("[INFO] training head...")
H = model.fit_generator(
    trainGen,
    steps_per_epoch=totalTrain // config.BATCH_SIZE,
    validation_data=valGen,
    validation_steps=totalVal // config.BATCH_SIZE,
    epochs=50)
```

```
# reset the testing generator and evaluate the network after
# fine-tuning just the network head
print("[INFO] evaluating after fine-tuning network head...")
testGen.reset()
predIdxs = model.predict_generator(testGen,
        steps=(totalTest // config.BATCH_SIZE) + 1)
predIdxs = np.argmax(predIdxs, axis=1)
print(classification_report(testGen.classes, predIdxs,
        target_names=testGen.class_indices.keys()))
plot_training(H, 50, config.WARMUP_PLOT_PATH)

# reset our data generators
trainGen.reset()
valGen.reset()

# now that the head FC layers have been trained/initialized, lets
# unfreeze the final set of CONV layers and make them trainable
for layer in baseModel.layers[15:]:
        layer.trainable = True
```

- train.py (6)

```
# loop over the layers in the model and show which ones are
trainable or not
for layer in baseModel.layers:
        print("{}: {}".format(layer, layer.trainable))

# for the changes to the model to take affect we need to
#recompile the model, this time using SGD with a *very*
small learning rate
print("[INFO] re-compiling model...")
opt = SGD(lr=1e-4, momentum=0.9)
model.compile(loss="categorical_crossentropy",
        optimizer=opt,metrics=["accuracy"])

# train the model again, this time fine-tuning *both* the
#final set of CONV layers along with our set of FC layers
H = model.fit_generator(
    trainGen,
    steps_per_epoch=totalTrain // config.BATCH_SIZE,
    validation_data=valGen,
    validation_steps=totalVal // config.BATCH_SIZE,
    epochs=20)
```

```
# reset the testing generator and then use our trained
# model to make predictions on the data
print("[INFO] evaluating after fine-tuning network...")
testGen.reset()
predIdxs = model.predict_generator(testGen,
        steps=(totalTest // config.BATCH_SIZE) + 1)
predIdxs = np.argmax(predIdxs, axis=1)
print(classification_report(testGen.classes, predIdxs,
        target_names=testGen.class_indices.keys()))
plot_training(H, 20, config.UNFROZEN_PLOT_PATH)

# serialize the model to disk
print("[INFO] serializing network...")
model.save(config.MODEL_PATH)
```

Transfer Learning (21) – Fine-tuning : Actual Practice – Foods classification (10)

- Execute result of “train.py”:

```
[INFO] training head...
W1001 15:01:45.382193  8600 deprecation.py:323] From C:\ProgramData\Anaconda3\envs\BGKim\lib\site-packages\tensorflow\python\ops\mat
d.py:1250: add_dispatch_support.<locals>.wrapper (from tensorflow.python.ops.array_ops) is deprecated and will be removed in a futur
sion.
Instructions for updating:
Use tf.where in 2.0, which has the same broadcast rule as np.where
Epoch 1/50
308/308 [==============================] - 135s 438ms/step - loss: 10.8853 - acc: 0.2781 - val_loss: 8.0759 - val_acc: 0.4501
Epoch 2/50
308/308 [==============================] - 120s 390ms/step - loss: 8.1763 - acc: 0.4297 - val_loss: 5.3619 - val_acc: 0.5989
Epoch 3/50
128/308 [===========>..................] - ETA: 58s - loss: 6.6144 - acc: 0.5042
```

```
<keras.layers.pooling.MaxPooling2D object at 0x000001E7703AF1D0>: False
<keras.layers.convolutional.Conv2D object at 0x000001E771357DD8>: False
<keras.layers.convolutional.Conv2D object at 0x000001E77137E160>: False
<keras.layers.convolutional.Conv2D object at 0x000001E77137EBE0>: False
<keras.layers.pooling.MaxPooling2D object at 0x000001E77139CE80>: False
<keras.layers.convolutional.Conv2D object at 0x000001E77139C390>: False
<keras.layers.convolutional.Conv2D object at 0x000001E7713B34E0>: False
<keras.layers.convolutional.Conv2D object at 0x000001E7713B3E10>: False
<keras.layers.pooling.MaxPooling2D object at 0x000001E7713CFC18>: False
<keras.layers.convolutional.Conv2D object at 0x000001E7713CFD68>: True
<keras.layers.convolutional.Conv2D object at 0x000001E7713E9860>: True
<keras.layers.convolutional.Conv2D object at 0x000001E7713F5CC0>: True
<keras.layers.pooling.MaxPooling2D object at 0x000001E771400FD0>: True
[INFO] re-compiling model
Epoch 1/20
308/308 [==============================] - 124s 402ms/step - loss: 0.8234 - acc: 0.7348 - val_loss: 0.7684 - val_acc: 0.7769
Epoch 2/20
308/308 [==============================] - 122s 396ms/step - loss: 0.7774 - acc: 0.7487 - val_loss: 0.7372 - val_acc: 0.7872
Epoch 3/20
308/308 [==============================] - 121s 391ms/step - loss: 0.7284 - acc: 0.7603 - val_loss: 0.6731 - val_acc: 0.7999
Epoch 4/20
308/308 [==============================] - 121s 392ms/step - loss: 0.6848 - acc: 0.7818 - val_loss: 0.6374 - val_acc: 0.8096
Epoch 5/20
308/308 [==============================] - 124s 402ms/step - loss: 0.6377 - acc: 0.7940 - val_loss: 0.6581 - val_acc: 0.8161
Epoch 6/20
308/308 [==============================] - 123s 399ms/step - loss: 0.6131 - acc: 0.8004 - val_loss: 0.6345 - val_acc: 0.8184
Epoch 7/20
308/308 [==============================] - 119s 386ms/step - loss: 0.5744 - acc: 0.8148 - val_loss: 0.6650 - val_acc: 0.8131
Epoch 8/20
308/308 [==============================] - 120s 389ms/step - loss: 0.5534 - acc: 0.8221 - val_loss: 0.6063 - val_acc: 0.8287
Epoch 9/20
308/308 [==============================] - 123s 398ms/step - loss: 0.5307 - acc: 0.8257 - val_loss: 0.5832 - val_acc: 0.8308
Epoch 10/20
308/308 [==============================] - 117s 380ms/step - loss: 0.5203 - acc: 0.8363 - val_loss: 0.6177 - val_acc: 0.8290
Epoch 11/20
308/308 [==============================] - 117s 378ms/step - loss: 0.4820 - acc: 0.8415 - val_loss: 0.5794 - val_acc: 0.8446
Epoch 12/20
308/308 [==============================] - 117s 381ms/step - loss: 0.4626 - acc: 0.8493 - val_loss: 0.5780 - val_acc: 0.8446
Epoch 13/20
308/308 [==============================] - 116s 376ms/step - loss: 0.4488 - acc: 0.8515 - val_loss: 0.6116 - val_acc: 0.8287
Epoch 14/20
308/308 [==============================] - 120s 389ms/step - loss: 0.4289 - acc: 0.8571 - val_loss: 0.5761 - val_acc: 0.8402
Epoch 15/20
308/308 [==============================] - 146s 475ms/step - loss: 0.4180 - acc: 0.8615 - val_loss: 0.5785 - val_acc: 0.8502
Epoch 16/20
308/308 [==============================] - 119s 387ms/step - loss: 0.3956 - acc: 0.8708 - val_loss: 0.5864 - val_acc: 0.8464
Epoch 17/20
308/308 [==============================] - 117s 379ms/step - loss: 0.3992 - acc: 0.8708 - val_loss: 0.5586 - val_acc: 0.8514
Epoch 18/20
308/308 [==============================] - 116s 376ms/step - loss: 0.3605 - acc: 0.8816 - val_loss: 0.6251 - val_acc: 0.8423
Epoch 19/20
308/308 [==============================] - 116s 378ms/step - loss: 0.3533 - acc: 0.8844 - val_loss: 0.5876 - val_acc: 0.8576
Epoch 20/20
```

```
308/308 [==============================] - 124s 402ms/step - loss: 0.3467 - acc: 0.8863 - val_loss: 0.5892 - val_acc: 0.8505
[INFO] evaluating after fine-tuning network...
               precision    recall  f1-score   support

        Bread       0.88      0.66      0.76       368
Dairy product       0.81      0.74      0.77       148
      Dessert       0.78      0.84      0.81       500
          Egg       0.82      0.83      0.82       335
   Fried food       0.84      0.82      0.83       287
         Meat       0.86      0.92      0.89       432
      Noodles       0.98      0.97      0.97       147
         Rice       0.88      0.96      0.92        96
      Seafood       0.88      0.90      0.89       303
         Soup       0.94      0.97      0.96       500
    Vegetable       0.95      0.95      0.95       231

     accuracy                           0.87      3347
    macro avg       0.87      0.87      0.87      3347
 weighted avg       0.87      0.87      0.86      3347

[INFO] serializing network...

(BGKim) C:\Users\vicl\practices\cnn\TransferLearning\Fine-Tuning>
(BGKim) C:\Users\vicl\practices\cnn\TransferLearning\Fine-Tuning>
```

- Evaluation module (predict.py) (1)

```
# import the necessary packages
from keras.models import load_model
from pyimagesearch import config
import numpy as np
import argparse
import imutils
import cv2

# construct the argument parser and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", type=str, required=True,
help="path to our input image")
args = vars(ap.parse_args())

# load the input image and then clone it so we can draw on it
later
image = cv2.imread(args["image"])
output = image.copy()
output = imutils.resize(output, width=400)

# our model was trained on RGB ordered images but OpenCV
represents
# images in BGR order, so swap the channels, and then resize to
# 224x224 (the input dimensions for VGG16)
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
image = cv2.resize(image, (224, 224))
```

```
# convert the image to a floating point data type and perform
mean
# subtraction
image = image.astype("float32")
mean = np.array([123.68, 116.779, 103.939][::1],
            dtype="float32")
image -= mean

# load the trained model from disk
print("[INFO] loading model...")
model = load_model(config.MODEL_PATH)

# pass the image through the network to obtain our
predictions
preds = model.predict(np.expand_dims(image, axis=0))[0]
i = np.argmax(preds)
label = config.CLASSES[i]

# draw the prediction on the output image
text = "{}: {:.2f}%".format(label, preds[i] * 100)
cv2.putText(output, text, (3, 20), cv2.FONT_HERSHEY_SIMPLEX,
            0.5,(0, 255, 0), 2)

# show the output image
cv2.imshow("Output", output)
cv2.waitKey(0)
```

- Let's run predict.py…!!!! And check on the verification results…..!!!!

```
(BGKim) C:₩Users₩vicl₩practices₩cnn₩TransferLearning₩Fine-Tuning>python predict.py --image
dataset₩evaluation₩Seafood₩8_102.jpg
```

Homewrok#2 Performance Comparison of Two Transfer learning schemes

- Content:
  - With the same dataset (one dataset), we want to compare the classification accuracy.
    - From two datasets, you can select one dataset
    - Implement same output classifier in each scheme.
    - Just training two scheme with one dataset
    - Just check on the accuracy.

- Submission:
  - Your technical report with your source code (compressed file)
  - Due: in a week from now.

Thank you for your attention.!!!
QnA

http://ivpl.sookmyung.ac.kr