# 101 class food Image

## zero-shot image classification

Li Yuan, advisor: Yuankai Huo

December 12, 2021

## 1 The Challenge and how I resolved it

Since our food image dataset has 101 classes, and each class has 1,000 images, so in total we have 101,000 images. Each image has over 500 by 500 resolution size. This is huge dataset for deep learning model and training process is very slow.

## 2 Interacting with CLIP

This is the first part of report that shows how to download and run CLIP models, and perform zero-shot image classifications on 101-class American food. We only show essential codes which is used to illustrate our purple, while other supplemental and helper codes are hidden in this notebook-style report.

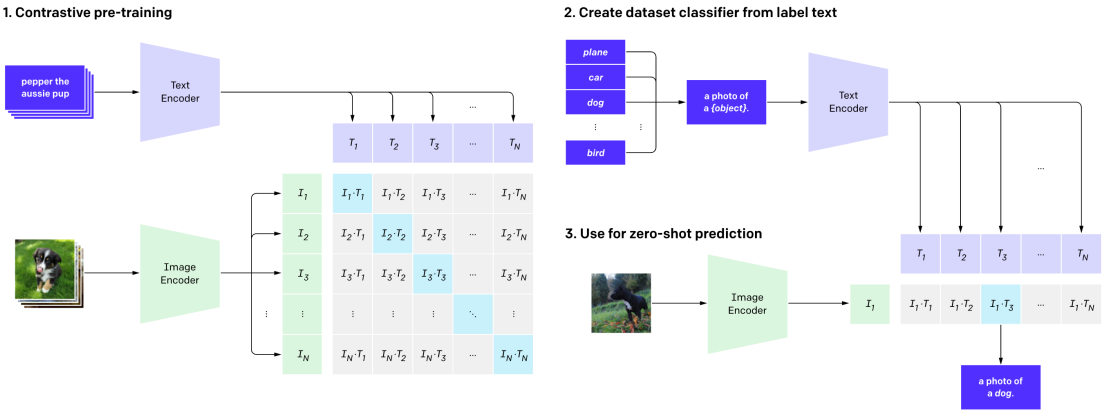
## 3 Introduction to CLIP (Contrastive Language–Image Pre-training)

> *"We're introducing a neural network called CLIP which efficiently learns visual concepts from natural language supervision. CLIP can be applied to any visual classification benchmark by simply providing the names of the visual categories to be recognized, similar to the "zero-shot" capabilities of GPT-2 and GPT-3."* https://openai.com/blog/clip/

## 4 Architecture of CLIP model

[3]:
```
from IPython.display import Image
Image(filename='../../codes/clip.png')
```

[3]:




*"CLIP pre-trains an image encoder and a text encoder to predict which images were paired with which texts in our dataset. We then use this behavior to turn CLIP into a zero-shot classifier. We convert all of a dataset's classes into captions such as "a photo of a dog" and predict the class of the caption CLIP estimates best pairs with a given image."* https://openai.com/blog/clip/

# 5 Mount my Google Drive

[1]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

## 5.1 Print out image and label folders

[4]:
```
% cd drive/MyDrive/Colab\ Notebooks/final/data/food-101/
% ls
```

```
/content/drive/MyDrive/Colab Notebooks/final/data/food-101
condacolab_install.log            meta/
small_val/
densenet-121_model_best_model.pt  README.txt
train/
images/                           resnet-50_model_best_model.pt
val/
license_agreement.txt             small_train/
```

# 6 Preparation for Colab

Make sure you're running a GPU runtime; if not, select "GPU" as the hardware accelerator in Runtime > Change Runtime Type in the menu. The next cells will install the `clip` package and its dependencies, and check if PyTorch 1.7.1 or later is installed.

## 6.1 Install conda in colab

[4]:
```
!pip install -q condacolab
import condacolab
condacolab.install()
```

```
 Downloading https://github.com/jaimergp/miniforge/releases/latest/download
/Mambaforge-colab-Linux-x86_64.sh...
 Installing...
 Adjusting configuration...
 Patching environment...
 Done in 0:00:23
 Restarting kernel...
```

[3]:
```
! conda --version
! which conda
```

```
conda 4.11.0
/usr/local/bin/conda
```

## 6.2 Install Pytorch 1.7 and its dependencies

[ ]:
```
! conda install --yes -c pytorch pytorch=1.7.1 torchvision cudatoolkit=11.0
! pip install ftfy regex tqdm
! pip install git+https://github.com/openai/CLIP.git
```

[6]:
```
import numpy as np
import torch

print("Torch version:", torch.__version__)

assert torch.__version__.split(".") >= ["1", "7", "1"], "PyTorch 1.7.1 or later␣
 ↪is required"
```

```
Torch version: 1.7.1
```

# 7 Delete impaired food images and count the remaining number in original data

When I directly trained this dataset, I found it threw an error to me that some images couldn't be opened. So I wrote this short script to detect all impaired image in training and validation set and deleted them. It also took a while to run due to the same chanllege that this is huge dataset.

[ ]:
```
all_folders = os.listdir("./images")
for folder in all_folders:
  all_filenames = os.listdir("./images/" + folder + "/")
  count = 0
  for i in all_filenames:
    try:
      test_image = Image.open("./images/" + folder + "/" + i)
    except:
      print(folder + " detects an impaired image: " + i)
      count = count + 1
      os.remove("./images/" + folder + "/" + i)
  print(folder + " found " + str(count) + " impaired images!")
  remaining = os.listdir("./images/" + folder + "/")
  print(folder + " has " + str(len(remaining)) + " images.")
  print("\n\n\n")
```

**After detecting and deleting, we found there is only one class, prime_rib, which contains impaired images that are unable to be opened by the image function. They are**

1. prime_rib detects an impaired image: 741587.jpg
2. prime_rib detects an impaired image: 348974.jpg
3. prime_rib detects an impaired image: 1953571.jpg
4. prime_rib detects an impaired image: 3595631.jpg
5. prime_rib detects an impaired image: 2597471.jpg

**prime_rib found 5 impaired images!**
**prime_rib has 995 images.**

## 8 Loading the model

`clip.available_models()` will list the names of available CLIP models.

[7]:
```
import clip

clip.available_models()
```

[7]:
```
['RN50', 'RN101', 'RN50x4', 'RN50x16', 'ViT-B/32', 'ViT-B/16']
```

[22]:
```
model, preprocess = clip.load("ViT-B/16")
model.cuda().eval()
input_resolution = model.visual.input_resolution
context_length = model.context_length
vocab_size = model.vocab_size

print("Model parameters:", f"{np.sum([int(np.prod(p.shape)) for p in model.
 ↪parameters()]):,}")
print("Input resolution:", input_resolution)
```

```
print("Context length:", context_length)
print("Vocab size:", vocab_size)
```

```
Model parameters: 149,620,737
Input resolution: 224
Context length: 77
Vocab size: 49408
```

As seen from this model parameters, this CLIP model is still like other deep CNN model with around 0.14 billion parameters. The input resolution is also like other pretrained model having 224 by 224, beyond image, this model also needs context text to predict labels.

## 9 Image Preprocessing

We resize the input images and center-crop them to conform with the image resolution that the model expects. Before doing so, we will normalize the pixel intensity using the dataset mean and standard deviation.

The second return value from `clip.load()` contains a torchvision `Transform` that performs this preprocessing.

[23]:
```
preprocess
```

[23]:
```
Compose(
    Resize(size=224, interpolation=PIL.Image.BICUBIC)
    CenterCrop(size=(224, 224))
    <function _convert_image_to_rgb at 0x7f78e6a87b90>
    ToTensor()
    Normalize(mean=(0.48145466, 0.4578275, 0.40821073), std=(0.26862954,
0.26130258, 0.27577711))
)
```

## 10 Text Preprocessing

We use a case-insensitive tokenizer, which can be invoked using `clip.tokenize()`. By default, the outputs are padded to become 77 tokens long, which is what the CLIP models expects.

[24]:
```
clip.tokenize("Hello World!")
```

[24]:
```
tensor([[49406,  3306,  1002,   256, 49407,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0,     0,     0,     0,
            0,     0,     0,     0,     0,     0,     0]])
```

## 11 Setting up input images and texts

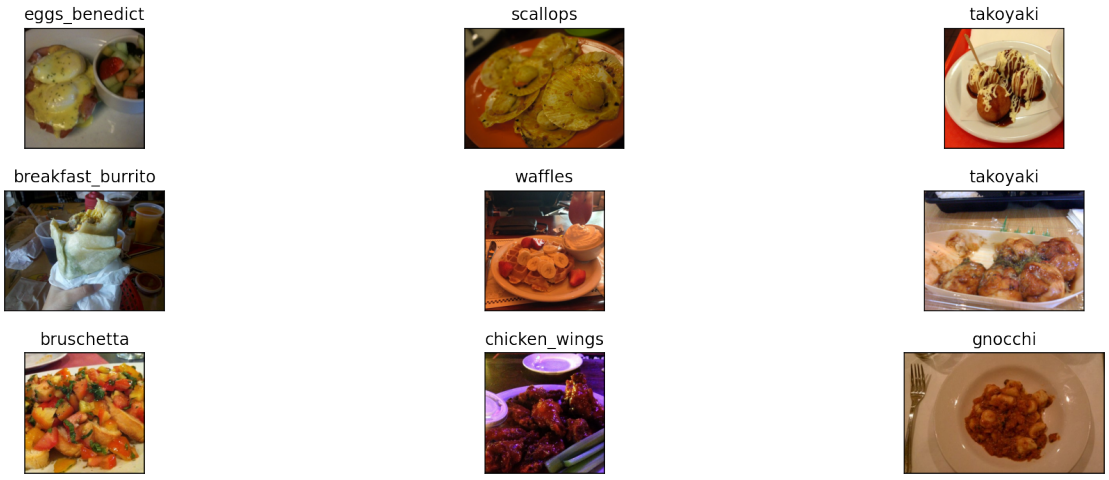
We are going to feed all 101,000 food images and setting up all 101 label descriptions to the model, then compute text feature and image feature.

The tokenizer is case-insensitive, and we can freely give any suitable textual descriptions.

[25]:
```
import os
import skimage
import IPython.display
import matplotlib.pyplot as plt
from PIL import Image
import numpy as np

from collections import OrderedDict
import torch

%matplotlib inline
%config InlineBackend.figure_format = 'retina'
```

## 12 Randomly Show 10 class food original image

[12]:
```
all_folders = os.listdir("./images")
random_index = np.random.randint(low=0, high=len(all_folders), size=9)
select_class = [all_folders[i] for i in random_index]
select_class

plt.figure(figsize=(16, 5))
count = 1

for filename in select_class:

  all_images = os.listdir(os.path.join("./images/" + filename))
  random_ind = np.random.randint(low=0, high=len(all_images), size=1)[0]
  image = Image.open(os.path.join("./images/" + filename + "/" +␣
 ↪all_images[random_ind])).convert("RGB")

  plt.subplot(3, 3, count)
  plt.imshow(image)
  plt.title(f"{filename}")
  plt.xticks([])
  plt.yticks([])
  count = count + 1

plt.tight_layout()
```




## 13 Use all 101 labels to generate text descriptions for model to pair, then compute text features

[26]:
```
labels = open('./meta/labels.txt', 'r')
labels = labels.readlines()
labels = [i.strip() for i in labels]

text_descriptions = [f"a photo of {label}, a type of food." for label in labels]
text_tokens = clip.tokenize(text_descriptions).cuda()


with torch.no_grad():
    text_features = model.encode_text(text_tokens).float()
    text_features /= text_features.norm(dim=-1, keepdim=True)
```

## 14 Convert all 101,000 food images into each 1 by 512 image features, them compute cosine similarity between text features and images feature, find the maximum one

[ ]:
```
all_folders = os.listdir("./images")
accuracy = []

for i in all_folders:
  all_png = os.listdir("./images/" + i)
  original_images = [Image.open("./images/" + i + "/" + j).convert("RGB") for j␣
 ↪in all_png]
```

```
  images = [preprocess(j) for j in original_images]
  image_input = torch.tensor(np.stack(images)).cuda()

  with torch.no_grad():
      image_features = model.encode_image(image_input).float()

  text_probs = (100.0 * image_features @ text_features.T).softmax(dim=-1)
  top_probs, top_labels = text_probs.cpu().topk(10, dim=-1)

  true_label = list(map(lambda x: x.replace(" ", "_").lower(), labels)).index(i)
  acc = np.sum((top_labels[:, 0] == true_label).tolist()) / top_labels.shape[0]
  accuracy.append(acc)
  print(i, "has", image_features.shape[0], "images with", "accuracy:", acc)
```

```
samosa has 1000 images with accuracy: 0.888
sashimi has 1000 images with accuracy: 0.889
spring_rolls has 1000 images with accuracy: 0.903
panna_cotta has 1000 images with accuracy: 0.777
greek_salad has 1000 images with accuracy: 0.87
foie_gras has 1000 images with accuracy: 0.675
tacos has 1000 images with accuracy: 0.741
pad_thai has 1000 images with accuracy: 0.957
ramen has 1000 images with accuracy: 0.76
pulled_pork_sandwich has 1000 images with accuracy: 0.922
bibimbap has 1000 images with accuracy: 0.93
beignets has 1000 images with accuracy: 0.919
crab_cakes has 1000 images with accuracy: 0.905
risotto has 1000 images with accuracy: 0.844
steak has 1000 images with accuracy: 0.391
frozen_yogurt has 1000 images with accuracy: 0.961
club_sandwich has 1000 images with accuracy: 0.929
carrot_cake has 1000 images with accuracy: 0.847
falafel has 1000 images with accuracy: 0.854
chicken_wings has 1000 images with accuracy: 0.934
chocolate_cake has 1000 images with accuracy: 0.828
tiramisu has 1000 images with accuracy: 0.817
spaghetti_bolognese has 1000 images with accuracy: 0.944
garlic_bread has 1000 images with accuracy: 0.825
scallops has 1000 images with accuracy: 0.772
edamame has 1000 images with accuracy: 0.993
pancakes has 1000 images with accuracy: 0.903
red_velvet_cake has 1000 images with accuracy: 0.918
deviled_eggs has 1000 images with accuracy: 0.935
peking_duck has 1000 images with accuracy: 0.889
guacamole has 1000 images with accuracy: 0.928
clam_chowder has 1000 images with accuracy: 0.908
croque_madame has 1000 images with accuracy: 0.889
```

```
french_onion_soup has 1000 images with accuracy: 0.891
beef_carpaccio has 1000 images with accuracy: 0.89
donuts has 1000 images with accuracy: 0.833
ravioli has 1000 images with accuracy: 0.687
spaghetti_carbonara has 1000 images with accuracy: 0.961
french_fries has 1000 images with accuracy: 0.92
shrimp_and_grits has 1000 images with accuracy: 0.917
dumplings has 1000 images with accuracy: 0.909
tuna_tartare has 1000 images with accuracy: 0.77
sushi has 1000 images with accuracy: 0.863
takoyaki has 1000 images with accuracy: 0.937
breakfast_burrito has 1000 images with accuracy: 0.836
macarons has 1000 images with accuracy: 0.958
waffles has 1000 images with accuracy: 0.901
seaweed_salad has 1000 images with accuracy: 0.972
cannoli has 1000 images with accuracy: 0.939
pizza has 1000 images with accuracy: 0.941
hot_and_sour_soup has 1000 images with accuracy: 0.922
prime_rib has 995 images with accuracy: 0.8753768844221106
ice_cream has 1000 images with accuracy: 0.623
pho has 1000 images with accuracy: 0.946
lobster_roll_sandwich has 1000 images with accuracy: 0.953
nachos has 1000 images with accuracy: 0.949
oysters has 1000 images with accuracy: 0.974
escargots has 1000 images with accuracy: 0.78
strawberry_shortcake has 1000 images with accuracy: 0.843
lobster_bisque has 1000 images with accuracy: 0.891
chicken_curry has 1000 images with accuracy: 0.82
bread_pudding has 1000 images with accuracy: 0.774
grilled_cheese_sandwich has 1000 images with accuracy: 0.799
baby_back_ribs has 1000 images with accuracy: 0.828
caprese_salad has 1000 images with accuracy: 0.887
pork_chop has 1000 images with accuracy: 0.798
beet_salad has 1000 images with accuracy: 0.763
paella has 1000 images with accuracy: 0.891
bruschetta has 1000 images with accuracy: 0.815
hummus has 1000 images with accuracy: 0.835
grilled_salmon has 1000 images with accuracy: 0.843
cheese_plate has 1000 images with accuracy: 0.945
filet_mignon has 1000 images with accuracy: 0.756
mussels has 1000 images with accuracy: 0.943
cheesecake has 1000 images with accuracy: 0.73
gyoza has 1000 images with accuracy: 0.796
chicken_quesadilla has 1000 images with accuracy: 0.896
huevos_rancheros has 1000 images with accuracy: 0.879
hot_dog has 1000 images with accuracy: 0.922
fish_and_chips has 1000 images with accuracy: 0.924
onion_rings has 1000 images with accuracy: 0.893
```

```
omelette has 1000 images with accuracy: 0.858
apple_pie has 1000 images with accuracy: 0.791
hamburger has 1000 images with accuracy: 0.867
beef_tartare has 1000 images with accuracy: 0.751
baklava has 1000 images with accuracy: 0.831
chocolate_mousse has 1000 images with accuracy: 0.711
miso_soup has 1000 images with accuracy: 0.861
gnocchi has 1000 images with accuracy: 0.782
eggs_benedict has 1000 images with accuracy: 0.932
caesar_salad has 1000 images with accuracy: 0.941
macaroni_and_cheese has 1000 images with accuracy: 0.819
french_toast has 1000 images with accuracy: 0.869
cup_cakes has 1000 images with accuracy: 0.741
poutine has 1000 images with accuracy: 0.916
lasagna has 1000 images with accuracy: 0.845
fried_calamari has 1000 images with accuracy: 0.904
churros has 1000 images with accuracy: 0.909
fried_rice has 1000 images with accuracy: 0.923
creme_brulee has 1000 images with accuracy: 0.928
ceviche has 1000 images with accuracy: 0.732
```

## 15 Conclusion

These are all accuracy for each class. As we can see, the accuracy is pretty high around 90% for some classes, this suggests this CLIP zero-shot model performs very well on these 101,000 huge image dataset. We have 101 labels for zero-shot to predict, this is a tough task, because the random guess for each image is less than 1%, but our CLIP model predict around 90% accuracy. This is amazing. Actually, CLIP architecture provides four models to use,

`'RN50', 'RN101', 'RN50x4', 'RN50x16', 'ViT-B/32', 'ViT-B/16'`.

**After trying each one, I found the `ViT-B/16` performs better than other models, so I chose this one to train for final result. `RN50` gives around 77% accuracy, and `RN101` gives around 79% accuracy while `ViT-B/16` gives around 83.82%.**

The challenge is that when I tried model `RN50x4` and `RN50x16`, the colab Pro gave me an error that `Out of CUDA memory` which means currently it can't run zero-shot models of these two. This is limitation of our hardware. I wish I can run all of the available models to compare their performance. I found these two models have much larger pretrain parameters like over 600MB when downloading them. This is also Deep Learning limitation because it requires a high-performance architecture.

# 101 food image pretrain classification

Li Yuan, advisor: Yuankai Huo

December 12, 2021

## 1 Transfer Learning In PyTorch

The goal of this second part is to implement pretrained models in Pytorch to perform image classification and test it out on 101-class food images. All the essential codes will be displayed in this notebook-style report and unnecessary codes will be hidden.

## 2 The Challenge and how I resolved it

Since our food image dataset has 101 classes, and each class has 1,000 images, so in total we have 101,000 images. Each image has over 500 by 500 pixel size. This is huge dataset for deep learning model and training process is very slow. In order to choose the best model architecture for this dataset, I shrink the dataset into each class training set has 200 images and each class validation set has 50 images. I trained different models on this subset and found the best model densenet-161, then I applied this model to the whole dataset.

## 3 Mount Google drive so that it can access images directly

[74]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Drive already mounted at /content/gdrive; to attempt to forcibly remount, call
drive.mount("/content/gdrive", force_remount=True).
```

## 4 Change working directory to image folder

[2]:
```
% cd gdrive/MyDrive/Colab\ Notebooks/final/data/food-101/
% ls
```

```
/content/gdrive/MyDrive/Colab Notebooks/final/data/food-101
condacolab_install.log            meta/
small_val/
densenet-121_model_best_model.pt  README.txt
train/
images/                           resnet-50_model_best_model.pt
```

```
val/
license_agreement.txt              small_train/
```

## 5 Install modules not already installed by Google Colab.

[ ]:
```
! pip install torch_utils
```

## 6 Import all needed packages

[63]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# import modules
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%config InlineBackend.figure_format = 'pdf'

import torch
from torch import cuda
import torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader, sampler
import torch.nn.functional as F
from torch_utils import AverageMeter
import math
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
from numpy import inf
import torchvision

from sklearn.model_selection import train_test_split
import os
from glob import glob
from torchvision import transforms
from torchvision import datasets
from torchvision import models
from torch import optim, cuda, Tensor
import tqdm

# Data science tools
import numpy as np

#import os
from shutil import copyfile, rmtree
import json
```

```
# Image manipulations
from PIL import Image
from timeit import default_timer as timer

# Visualizations
import matplotlib.pyplot as plt
#plt.rcParams['font.size'] = 14

import warnings
warnings.filterwarnings('ignore', category=FutureWarning)
```

## 7 Create training and test dataset

The downloaded image data don't contain training and test set but provides training and test index. So I wrote a short script to split the whole data into training and test set by copyfile function. It took me the whole day to run this splitting process. So, again, the biggest chanllege is that this data is so big that reading, processing and training are very time consuming. Thanks to this colab which can run all these stuff continuously.

[ ]:
```
test = open('./meta/test.json')
test_id = json.load(test)

train = open('./meta/train.json')
train_id = json.load(train)
```

[ ]:
```
rmtree("./val", ignore_errors=True)
rmtree("./train", ignore_errors=True)

os.mkdir("./val")
os.mkdir("./train")

for name, ids in test_id.items():
  all_ids = os.listdir("./images/" + name)
  pids = list(map(lambda x: x.split("/")[1], ids))
  for i in all_ids:
    id = os.path.splitext(i)[0]
    if id in pids:
      if name not in os.listdir("./val"):
        os.mkdir("./val/" + name)
      copyfile("./images/" + name + "/" + i, "./val/" + name + "/" + i)
    else:
      if name not in os.listdir("./train"):
        os.mkdir("./train/" + name)
      copyfile("./images/" + name + "/" + i, "./train/" + name + "/" + i)
```

## 8 Delete impaired images and count number of images in each class

When I directly trained this dataset, I found it threw an error to me that some images couldn't be opened. So I wrote this short script to detect all impaired image in training and validation set and deleted them. It also took a while to run due to the same chanllege that this is huge dataset.

### 8.1 Training part

[ ]:
```
all_folders = os.listdir("./train")
for folder in all_folders:
  all_filenames = os.listdir("./train/" + folder + "/")
  count = 0
  for i in all_filenames:
    try:
      test_image = Image.open("./train/" + folder + "/" + i)
    except:
      print(folder + " detects an impaired image: " + i)
      count = count + 1
      os.remove("./train/" + folder + "/" + i)
  print(folder + " found " + str(count) + " impaired images!")
  remaining = os.listdir("./train/" + folder + "/")
  print(folder + " has " + str(len(remaining)) + " images.")
  print("\n\n\n")
```

### 8.2 Validation part

[ ]:
```
all_folders = os.listdir("./val")
for folder in all_folders:
  all_filenames = os.listdir("./val/" + folder + "/")
  count = 0
  for i in all_filenames:
    try:
      test_image = Image.open("./val/" + folder + "/" + i)
    except:
      print(folder + " detects an impaired image: " + i)
      count = count + 1
      os.remove("./val/" + folder + "/" + i)
  print(folder + " found " + str(count) + " impaired images!")
  remaining = os.listdir("./val/" + folder + "/")
  print(folder + " has " + str(len(remaining)) + " images.")
  print("\n\n\n")
```

**After detecting and deleting, we found there is only one class, prime_rib, which contains impaired images that are unable to be opened by the image function. They are**

1. prime_rib detects an impaired image: 741587.jpg
2. prime_rib detects an impaired image: 348974.jpg

3. prime_rib detects an impaired image: 1953571.jpg
4. prime_rib detects an impaired image: 3595631.jpg
5. prime_rib detects an impaired image: 2597471.jpg

**prime_rib found 5 impaired images!**
**prime_rib has 995 images.**

# 9 Random choose 200 food images of each class and 50 validation food images

As always, this big data chanllege prevented me from trying different models, because it took me for half of day to run one model. So, I decided to shrink our data into 200 for each class training set and 50 for each class validation set. I wrote a short script to reach this goal.

## 9.1 Training part

[ ]:
```
rmtree("./small_train", ignore_errors=True)
os.mkdir("./small_train")

all_folders = os.listdir("./train")
for folder in all_folders:
  all_filenames = os.listdir("./train/" + folder + "/")
  seed = np.random.randint(low=0, high=len(all_filenames), size=200)
  selected_filenames = [all_filenames[f] for f in seed]
  if folder not in os.listdir("./small_train"):
    os.mkdir("./small_train/" + folder)
  for each in selected_filenames:
    copyfile("./train/" + folder + "/" + each, "./small_train/" + folder + "/"␣
 ↪+ each)
```

## 9.2 Validation part

[ ]:
```
rmtree("./small_val", ignore_errors=True)
os.mkdir("./small_val")

all_folders = os.listdir("./val")
for folder in all_folders:
  all_filenames = os.listdir("./val/" + folder + "/")
  seed = np.random.randint(low=0, high=len(all_filenames), size=50)
  selected_filenames = [all_filenames[f] for f in seed]
  if folder not in os.listdir("./small_val"):
    os.mkdir("./small_val/" + folder)
  for each in selected_filenames:
    copyfile("./val/" + folder + "/" + each, "./small_val/" + folder + "/" +␣
 ↪each)
```

## 10 Try different pretrained model and report their results

1. resnet50(pretrained=True)

   **Use image augmentation**

```
model.fc = nn.Sequential(nn.Dropout(p=0.6),
                         nn.Linear(num_ftrs, 101),
                         nn.LogSoftmax(dim=1))
```

   We added one dropout layer and changed the final layer by adding a softmax layer.
   **We ended up with 29 epoches, 2.0 loss, 51.33% accuracy.**

2. vgg16(pretrained=True)

   **Use image augmentation**

```
model.classifier[6] = nn.Linear(4096, 101, bias=True)
```

   **We ended up with 29 epoches, 1.99 loss, 49.09% accuracy.**

3. efficientnet_b7(pretrained=True)

   **Use image augmentation**

```
model.classifier = nn.Sequential(nn.Linear(in_features=num_ftrsout_features=num_ftrs//2, bias=
                     nn.ReLU(inplace=True),
                     nn.Dropout(p=0.5, inplace=False),
                     nn.Linear(in_features=num_ftrs//2, out_features=101, bias=True),
                     nn.LogSoftmax(dim=1))

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4, betas=(0.7, 0.8))
```

   **We ended up with 29 epoches, 2.12 loss, 46.85% accuracy.**

4. regnet_x_800mf(pretrained=True)

   **Don't use image augmentation since we have enough data.**

```
model.fc = nn.Sequential(
                     nn.Linear(in_features=num_ftrs, out_features=101, bias=True)
                     )

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.999))
```

   **We ended up with 25 epoches, 1.65 loss, 57.89% accuracy.**

5. vgg16(pretrained=True)

**Use image augmentation**

```
model.classifier = nn.Sequential(
                    nn.Linear(in_features=25088, out_features=4096, bias=True),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=0.5, inplace=False),
                    nn.Linear(in_features=4096, out_features=4096, bias=True),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=0.5, inplace=False),
                    nn.Linear(in_features=4096, out_features=1000, bias=True),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=0.5, inplace=False),
                    nn.Linear(in_features=1000, out_features=500, bias=True),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=0.5, inplace=False),
                    nn.Linear(in_features=500, out_features=101, bias=True),
                    nn.ReLU(inplace=True),
                    nn.Dropout(p=0.5, inplace=False))
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
```

**We ended up with 29 epoches, 3.72 loss, 28.06% accuracy.**

6. regnet_x_800mf(pretrained=True)

**Don't use image augmentation since we have enough data.**

```
model.fc = nn.Sequential(
                    nn.Linear(in_features=num_ftrs, out_features=101, bias=True)
                    )

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.999))
```

**We ended up with 27 epoches, 1.89 loss, 52.11% accuracy.**

7. densenet161(pretrained=True)

**Don't use image augmentation since we have enough data.**

```
model.classifier = nn.Sequential(
                    nn.Linear(in_features=num_ftrs, out_features=101, bias=True))

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.999))
```

**We ended up with 19 epoches, 1.27 loss, 65.31% accuracy.**

## 11 After comparing all of them on a small subset, we finally found the highest accuracy model, densenet-161 without using image augmentation ending up with 65.11% accuracy on all validation set by using one final linear layer.

## 12 Use Densenet-161 for the 101 class American food image data

The convolutional network is better than the fully-connected network. However, the images we've used in CIFAR-10 are easy and the image sizes are small. If we need to do a more complicated classification task with high-resolution images, we may use a more deep network model. Here, we will use the pretrain densenet-161 model for this huge size image data.

[45]:
```
# define paths and parameters
traindir = f"./train"
validdir = f"./val"

save_file_name = f'densenet161-transfer.pt'
checkpoint_path = f'densenet161-transfer.pth'

# Change to fit hardware
batch_size = 50
```

For deep learning, data augmentation is significant for increasing the dataset and improving the performance. We will define crop, rotation, color jitter, and flip before we load images in our network.

[46]:
```
# define transforms
# Image transformations
image_transforms = {
    # Train uses data augmentation
    'train':
        transforms.Compose([
            # You can use transforms.RandomResizedCrop() for crop
            # You can use transforms.RandomRotation() for rotation
            # You can use transforms.ColorJitter() for colorjitter
            # You can use transforms.RandomHorizontalFlip() for flip
            ######################
            ### YOUR CODE HERE###
            ######################
            transforms.RandomResizedCrop(224),
            #transforms.RandomHorizontalFlip(),
            #transforms.ColorJitter(),
            #transforms.RandomRotation(degrees=30),
            ####################
            ### YOUR CODE END###
            ####################
            transforms.CenterCrop(size=224),  # Image net standards
            transforms.ToTensor(),
```

```
            transforms.Normalize([0.485, 0.456, 0.406],
                                 [0.229, 0.224, 0.225])  # Imagenet standards
        ]),
    # Validation does not use augmentation
    'valid':
        transforms.Compose([
            transforms.Resize(size=256),
            transforms.CenterCrop(size=224),
            transforms.ToTensor(),
            transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
        ])
}
```

Let's take a look at the data augmentation we've defined. After the data augmentation, the image values might be out of the range. Therefore, when we print the images, we may clip the value.

[64]:
```
# take a look at data augmentation

def imshow_tensor(image, ax=None, title=None):
    """Imshow for Tensor."""

    if ax is None:
        fig, ax = plt.subplots()

    # Set the color channel as the third dimension
    image = image.numpy().transpose((1, 2, 0))

    # Reverse the preprocessing steps
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    image = std * image + mean

    # Clip the image pixel values
    # You may use np.clip() to clip the value between 0 to 1.
    ######################
    ### YOUR CODE HERE###
    ######################
    image = np.clip(image, 0, 1)
    ####################
    ### YOUR CODE END###
    ####################
    ax.imshow(image)
    plt.axis('off')

    return ax, image

food = os.listdir("./small_train/")
```

```
random_food = food[np.random.randint(0, len(food))]
pics = os.listdir("./small_train/" + random_food)
random_pics = pics[np.random.randint(0, len(pics))]

ex_img = Image.open('./small_train/' + random_food + "/" + random_pics)

t = image_transforms['train']
plt.figure(figsize=(24, 24))

for i in range(4):
    ax = plt.subplot(2, 2, i + 1)
    _ = imshow_tensor(t(ex_img), ax=ax)

plt.tight_layout();
plt.show();
```

Now for each original image, we can obtain several processed images. Now we can load the grocery data here. Since we load all the images from folders, and we need to extract the labels from folder names, the dataloader might be different. Then, we can check all the classes in our new dataset.

[48]:
```
# Datasets from folders
data = {
    'train':
    datasets.ImageFolder(root=traindir, transform=image_transforms['train']),
    'valid':
    ######################
    ### YOUR CODE HERE###
    ######################
    datasets.ImageFolder(root=validdir, transform=image_transforms['valid'])
    ######################
    ### YOUR CODE END###
    ######################
}

# Dataloader iterators, make sure to shuffle
# You can use the same dataloader in HW3
######################
### YOUR CODE HERE###
######################
dataloaders = {
    'train': DataLoader(data['train'], batch_size = batch_size, shuffle=True,␣
 ↪num_workers=10),
    'valid': DataLoader(data['valid'], batch_size = batch_size, shuffle=True,␣
 ↪num_workers=10)
}
####################
### YOUR CODE END###
####################

# Iterate through the dataloader once
trainiter = iter(dataloaders['train'])
validationiter = iter(dataloaders['valid'])

categories = []
for d in os.listdir(traindir):
    categories.append(d)

n_classes = len(categories)
print(f'There are {n_classes} different classes.')
```

```
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481:
```

```
UserWarning: This DataLoader will create 10 worker processes in total. Our
suggested max number of worker in current system is 4, which is smaller than
what this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))

There are 101 different classes.
```

Here, we may define our model with the pretrianed resnet-50. For the pretrained model, the output channels might match the original dataset. Therefore, we need to change the last block of network specially for our dataset. Because we use pretrain model, we may freeze all the layers that related to the features learning. Then, our model can only learn the feature classifier with the last block we've defined.

[49]:
```
# Define the network with pretrained models.resnet50
######################
### YOUR CODE HERE###
######################
model = models.densenet161(pretrained=True)
####################
### YOUR CODE END###
####################


# Freeze model weights
# You need to go throught all the parameters in model.parameters()
# You need to set "requires_grad" to "False" for all parameters
######################
### YOUR CODE HERE###
######################
for param in model.parameters():
    param.requires_grad = False
####################
### YOUR CODE END###
####################

#print(model)

# You may get the number of the features from the feature layer of the␣
 ↪pretrained network
# You can use model.fc.in_features to get the feature number
######################
### YOUR CODE HERE###
######################
num_ftrs = model.classifier.in_features
#print("Last layer input feature: ", num_ftrs)
####################
### YOUR CODE END###
```

```
####################


model.classifier = nn.Sequential(
                      # Define the last block of the nework for our dataset.
                      # This block may have two linear layers, one dropout␣
 ↪layer, and one softmax layer.
                      # You may design your own classifier with a discription.
                      # You can use nn.Linear() as your linear layers
                      # You can use nn.ReLU() as your relu layer
                      # You can use nn.Dropout() as your dropout layer
                      # You can use nn.LogSoftmax() as your softmax layer
                      ######################
                      ### YOUR CODE HERE###
                      ######################
                      nn.Linear(in_features=num_ftrs, out_features=101,␣
 ↪bias=True)
                      )
                      ####################
                      ### YOUR CODE END###
                      ####################


model.classifier

total_params = sum(p.numel() for p in model.parameters())
print("\n")
print(f'{total_params:,} total parameters.')
total_trainable_params = sum(
    p.numel() for p in model.parameters() if p.requires_grad)
print(f'{total_trainable_params:,} training parameters.')
```

[49]:
```
Sequential(
  (0): Linear(in_features=2208, out_features=101, bias=True)
)
```

```
26,695,109 total parameters.
223,109 training parameters.
```

Cuda is Compute Unified Device Architecture, which can achieve parallel computing. It will improve your learning speed in your parameter update by using GPU rather than CPU.

[50]:
```
# Check whether there is a gpu for cuda
train_on_gpu = cuda.is_available()
print(f'Train on gpu: {train_on_gpu}')

# Number of gpus
```

```
if train_on_gpu:
    gpu_count = cuda.device_count()
    print(f'{gpu_count} gpus detected.')
    if gpu_count > 1:
        multi_gpu = True
    else:
        multi_gpu = False
else:
    multi_gpu = False
print(train_on_gpu,multi_gpu)

if train_on_gpu:
    model = model.to('cuda')
```

```
Train on gpu: True
1 gpus detected.
True False
```

[ ]:
```
model.class_to_idx = data['train'].class_to_idx
model.idx_to_class = {
    idx: class_
    for class_, idx in model.class_to_idx.items()
}

list(model.idx_to_class.items())


# Set up your criterion and optimizer
# You can use nn.CrossEntropyLoss() as your critenrion
# You can use optim.Adam() as your optimizer with a reasonable paramter setting

######################
### YOUR CODE HERE###
######################
criterion = nn.CrossEntropyLoss()
#optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
optimizer = optim.Adam(model.parameters(), lr=1e-3, betas=(0.9, 0.999))
####################
### YOUR CODE END###
####################

for p in optimizer.param_groups[0]['params']:
    if p.requires_grad:
        print(p.shape)
```

Well down! Now we can start our training process here.

[52]:
```
# You can use your train function in HW3
def train(model,
          criterion,
          optimizer,
          train_loader,
          valid_loader,
          save_file_name,
          max_epochs_stop=3,
          n_epochs=10,
          print_every=1):
    """Train a PyTorch Model

    Params
    --------
        model (PyTorch model): cnn to train
        criterion (PyTorch loss): objective to minimize
        optimizer (PyTorch optimizier): optimizer to compute gradients of model␣
↪parameters
        train_loader (PyTorch dataloader): training dataloader to iterate␣
↪through
        valid_loader (PyTorch dataloader): validation dataloader used for early␣
↪stopping
        save_file_name (str ending in '.pt'): file path to save the model state␣
↪dict
        max_epochs_stop (int): maximum number of epochs with no improvement in␣
↪validation loss for early stopping
        n_epochs (int): maximum number of training epochs
        print_every (int): frequency of epochs to print training stats

    Returns
    --------
        model (PyTorch model): trained cnn with best weights
        history (DataFrame): history of train and validation loss and accuracy
    """

    # Early stopping intialization
    epochs_no_improve = 0
    valid_loss_min = np.Inf

    valid_max_acc = 0
    history = []

    # Number of epochs already trained (if using loaded in model weights)
    try:
        print(f'Model has been trained for: {model.epochs} epochs.\n')
    except:
        model.epochs = 0
```

```
        print(f'Starting Training from Scratch.\n')

    overall_start = timer()

    # Main loop
    for epoch in range(n_epochs):

        # keep track of training and validation loss each epoch

        train_loss = 0.0
        valid_loss = 0.0

        train_acc = 0
        valid_acc = 0


        # Set to training

        model.train()

        start = timer()

        # Training loop
        for ii, (data, target) in enumerate(train_loader):

            # Tensors to gpu

            if train_on_gpu:
                model = model.cuda()
                data, target = data.cuda(), target.cuda()


            # Clear gradients
            optimizer.zero_grad()

            # Get your output from your model

            model = model.float()
            output = model(data.float())

            # Loss and backpropagation of gradients

            loss = criterion(output, target.long())
            loss.backward()

            # Update the parameters
            ######################
```

```
            ### YOUR CODE HERE###
            ######################
            optimizer.step()
            ######################
            ### YOUR CODE END ###
            ######################

            # Track train loss by multiplying average loss by number of␣
↪examples in batch

            train_loss += loss.item() * data.size(0)



            # Calculate accuracy by finding max log probability

            _, pred = torch.max(output, dim=1)
            correct_tensor = pred.eq(target.data.view_as(pred))


            # Need to convert correct tensor from int to float to average

            accuracy = torch.mean(correct_tensor.type(torch.FloatTensor))


            # Multiply average accuracy times the number of examples in batch

            train_acc += accuracy.item() * data.size(0)


            # Track training progress
            print(
                f'Epoch: {epoch}\t{100 * (ii + 1) / len(train_loader):.2f}%␣
↪complete. {timer() - start:.2f} seconds elapsed in epoch.',
                end='\r')

        # After training loops ends, start validation
        else:
            model.epochs += 1

            # Don't need to keep track of gradients
            with torch.no_grad():

                # Set to evaluation mode

                model.eval()
```

```
            # Validation loop
            for data, target in valid_loader:
                # Tensors to gpu

                if train_on_gpu:
                    model = model.cuda()
                    data, target = data.cuda(), target.cuda()

                # Forward pass
                model = model.float()
                output = model(data.float())

                # Validation loss
                loss = criterion(output, target.long())

                # Multiply average loss times the number of examples in␣
↪batch
                valid_loss += loss.item() * data.size(0)

                # Calculate validation accuracy
                _, pred = torch.max(output, dim=1)
                correct_tensor = pred.eq(target.data.view_as(pred))
                accuracy = torch.mean(
                    correct_tensor.type(torch.FloatTensor))

                # Multiply average accuracy times the number of examples
                valid_acc += accuracy.item() * data.size(0)


            # Calculate average losses
            train_loss = train_loss / len(train_loader.dataset)
            valid_loss = valid_loss / len(valid_loader.dataset)


            # Calculate average accuracy
            train_acc = train_acc / len(train_loader.dataset)
            valid_acc = valid_acc / len(valid_loader.dataset)
            history.append([train_loss, valid_loss, train_acc, valid_acc])

            # Print training and validation results
            if (epoch + 1) % print_every == 0:
                print(
                    f'\nEpoch: {epoch} \tTraining Loss: {train_loss:.4f}␣
↪\tValidation Loss: {valid_loss:.4f}'
                )
                print(
```

```
                        f'\t\tTraining Accuracy: {100 * train_acc:.2f}%\t␣
↪Validation Accuracy: {100 * valid_acc:.2f}%'
                    )

                # Save the model if validation loss decreases
                if valid_loss < valid_loss_min:
                    # Save model
                    torch.save(model.state_dict(), save_file_name)
                    # Track improvement
                    epochs_no_improve = 0
                    valid_loss_min = valid_loss
                    valid_best_acc = valid_acc
                    best_epoch = epoch

                # Otherwise increment count of epochs with no improvement
                else:
                    epochs_no_improve += 1
                    # Trigger early stopping
                    if epochs_no_improve >= max_epochs_stop:
                        print(
                            f'\nEarly Stopping! Total epochs: {epoch}. Best␣
↪epoch: {best_epoch} with loss: {valid_loss_min:.2f} and acc: {100 *␣
↪valid_acc:.2f}%'
                        )
                        total_time = timer() - overall_start
                        print(
                            f'{total_time:.2f} total seconds elapsed.␣
↪{total_time / (epoch+1):.2f} seconds per epoch.'
                        )

                        # Load the best state dict
                        model.load_state_dict(torch.load(save_file_name))

                        # Attach the optimizer
                        model.optimizer = optimizer

                        # Format history
                        history = pd.DataFrame(
                            history,
                            columns=[
                                'train_loss', 'valid_loss', 'train_acc',
                                'valid_acc'
                            ])
                        return model, history

    # Attach the optimizer
    model.optimizer = optimizer
```

```
    # Record overall time and print out stats
    total_time = timer() - overall_start
    print(
        f'\nBest epoch: {best_epoch} with loss: {valid_loss_min:.2f} and acc:␣
↪{100 * valid_best_acc:.2f}%'
    )
    print(
        f'{total_time:.2f} total seconds elapsed. {total_time / (epoch+1):.2f}␣
↪seconds per epoch.'
    )
    # Format history
    history = pd.DataFrame(
        history,
        columns=['train_loss', 'valid_loss', 'train_acc', 'valid_acc'])
    return model, history
```

[53]:
```
from timeit import default_timer as timer
save_file_name = f'resnet-50_model_best_model.pt'
train_on_gpu = cuda.is_available()

model, history = train(model,
    criterion,
    optimizer,
    dataloaders['train'],
    dataloaders['valid'],
    save_file_name=save_file_name,
    max_epochs_stop=10,
    n_epochs=30,
    print_every=1)
```

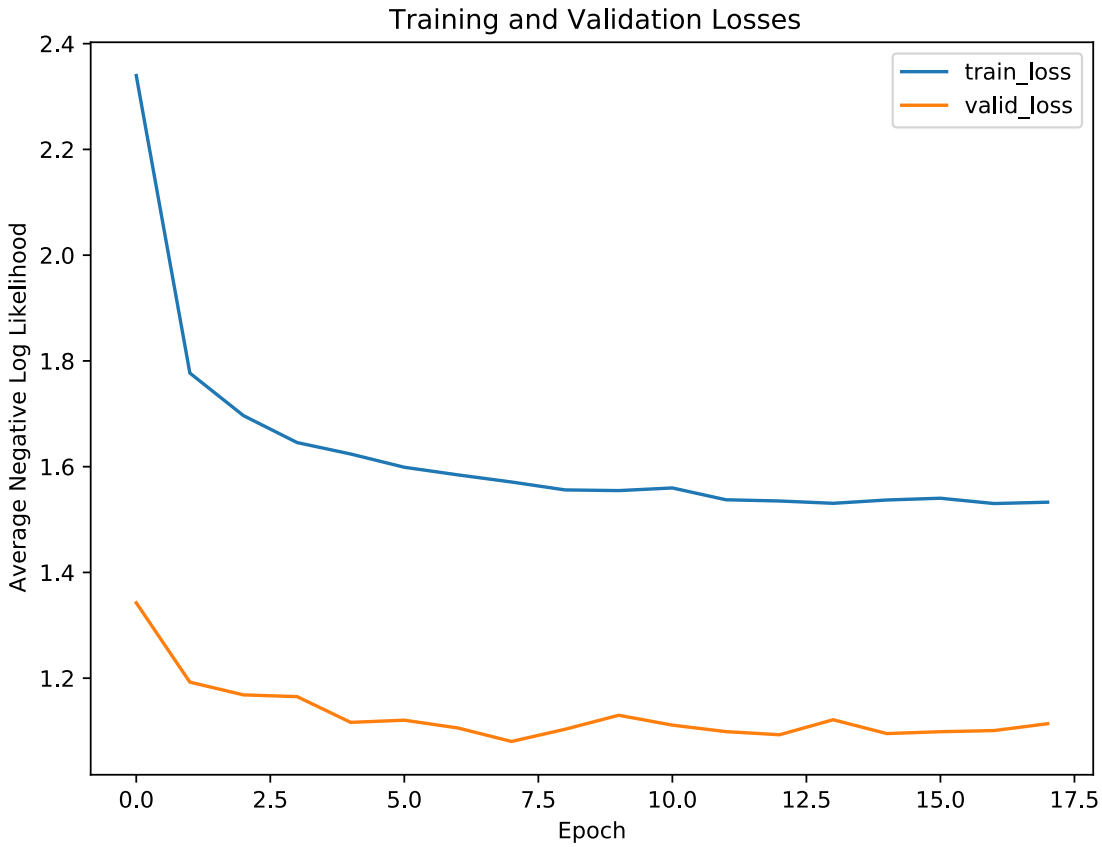
```
Starting Training from Scratch.


/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:481:
UserWarning: This DataLoader will create 10 worker processes in total. Our
suggested max number of worker in current system is 4, which is smaller than
what this DataLoader is going to create. Please be aware that excessive worker
creation might get DataLoader running slow or even freeze, lower the worker
number to avoid potential slowness/freeze if necessary.
  cpuset_checked))


Epoch: 0        Training Loss: 2.3397   Validation Loss: 1.3425
                Training Accuracy: 45.89%        Validation Accuracy: 64.99%

Epoch: 1        Training Loss: 1.7770   Validation Loss: 1.1925
                Training Accuracy: 56.23%        Validation Accuracy: 68.18%
```

```
Epoch: 2        Training Loss: 1.6965   Validation Loss: 1.1684
                Training Accuracy: 57.81%        Validation Accuracy: 68.63%

Epoch: 3        Training Loss: 1.6456   Validation Loss: 1.1651
                Training Accuracy: 58.86%        Validation Accuracy: 68.88%

Epoch: 4        Training Loss: 1.6240   Validation Loss: 1.1164
                Training Accuracy: 59.41%        Validation Accuracy: 69.90%

Epoch: 5        Training Loss: 1.5989   Validation Loss: 1.1205
                Training Accuracy: 59.92%        Validation Accuracy: 69.73%

Epoch: 6        Training Loss: 1.5845   Validation Loss: 1.1058
                Training Accuracy: 60.36%        Validation Accuracy: 70.17%

Epoch: 7        Training Loss: 1.5710   Validation Loss: 1.0803
                Training Accuracy: 60.63%        Validation Accuracy: 71.12%

Epoch: 8        Training Loss: 1.5560   Validation Loss: 1.1034
                Training Accuracy: 61.27%        Validation Accuracy: 70.36%

Epoch: 9        Training Loss: 1.5547   Validation Loss: 1.1299
                Training Accuracy: 60.96%        Validation Accuracy: 69.73%

Epoch: 10       Training Loss: 1.5597   Validation Loss: 1.1112
                Training Accuracy: 61.02%        Validation Accuracy: 70.12%

Epoch: 11       Training Loss: 1.5374   Validation Loss: 1.0989
                Training Accuracy: 61.47%        Validation Accuracy: 70.34%

Epoch: 12       Training Loss: 1.5351   Validation Loss: 1.0930
                Training Accuracy: 61.52%        Validation Accuracy: 70.61%

Epoch: 13       Training Loss: 1.5307   Validation Loss: 1.1214
                Training Accuracy: 61.68%        Validation Accuracy: 70.16%

Epoch: 14       Training Loss: 1.5370   Validation Loss: 1.0952
                Training Accuracy: 61.68%        Validation Accuracy: 70.48%

Epoch: 15       Training Loss: 1.5403   Validation Loss: 1.0988
                Training Accuracy: 61.58%        Validation Accuracy: 70.58%

Epoch: 16       Training Loss: 1.5303   Validation Loss: 1.1009
                Training Accuracy: 61.55%        Validation Accuracy: 70.57%

Epoch: 17       Training Loss: 1.5328   Validation Loss: 1.1139
                Training Accuracy: 61.63%        Validation Accuracy: 70.09%
```

```
Early Stopping! Total epochs: 17. Best epoch: 7 with loss: 1.08 and acc: 70.09%
11430.60 total seconds elapsed. 635.03 seconds per epoch.
```

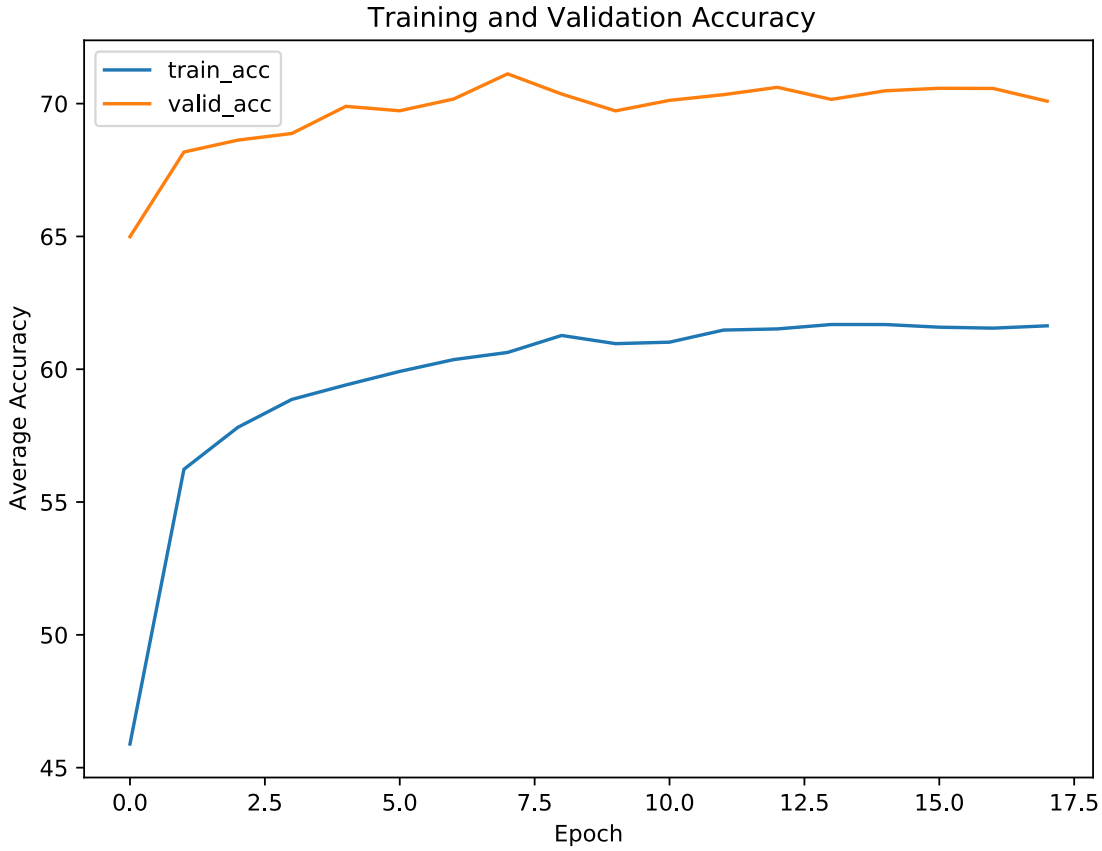
Let's check the results

[68]:
```
plt.figure(figsize=(8, 6));
for c in ['train_loss', 'valid_loss']:
    plt.plot(
        history[c], label=c)
plt.legend();
plt.xlabel('Epoch')
plt.ylabel('Average Negative Log Likelihood')
plt.title('Training and Validation Losses')
plt.show();
```




[69]:
```
plt.figure(figsize=(8, 6));
for c in ['train_acc', 'valid_acc']:
    plt.plot(
        100 * history[c], label=c)
plt.legend();
plt.xlabel('Epoch')
plt.ylabel('Average Accuracy')
```

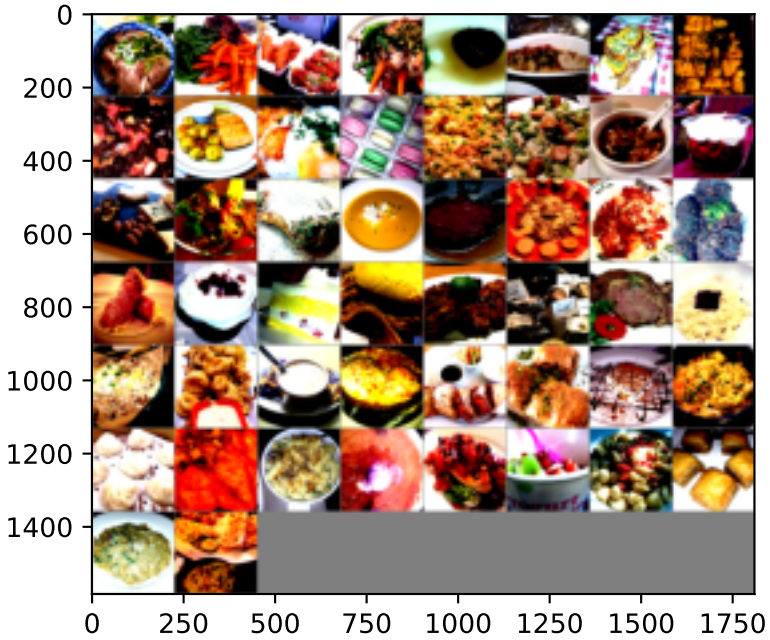
```
plt.title('Training and Validation Accuracy')
plt.show();
```




[71]:
```
# functions to show an image
def imshow(img):
    img = img / 2 + 0.5     # unnormalize
    npimg = img.numpy()
    npimg = npimg.transpose(1,2,0)

    plt.imshow(npimg)
    plt.show()


#dataiter = iter(validationiter)
# get some random training images
# you may use .next() to get the next iteration of validation dataloader
######################
### YOUR CODE HERE ###
######################
images, labels = validationiter.next()
```

```
######################
### YOUR CODE END ###
######################

# Get the prediction of images by using your model.
######################
### YOUR CODE HERE###
######################
outputs = model(images.cuda().float())
_, predicted = torch.max(outputs, 1)
######################
### YOUR CODE END###
######################

# print images
imshow(torchvision.utils.make_grid(images))
print('GroundTruth: ', ' '.join('%5s' % categories[labels[j].long()] for j in␣
 ↪range(batch_size)))
print('Prediction: ', ' '.join('%5s' % categories[predicted[j].long()] for j in␣
 ↪range(batch_size)))
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```




```
GroundTruth:  dumplings prime_rib paella sushi mussels fried_calamari
red_velvet_cake garlic_bread fried_calamari caesar_salad bread_pudding gyoza
```

```
french_fries pad_thai peking_duck cheesecake beignets bibimbap poutine
beef_carpaccio eggs_benedict sashimi chocolate_mousse spaghetti_bolognese
poutine chicken_wings cheesecake cup_cakes sushi ceviche eggs_benedict hot_dog
beet_salad garlic_bread frozen_yogurt macaroni_and_cheese pancakes samosa
lobster_roll_sandwich beef_tartare creme_brulee caesar_salad donuts
shrimp_and_grits tacos onion_rings grilled_salmon samosa hot_dog risotto
Prediction:  dumplings mussels paella nachos mussels fried_calamari
red_velvet_cake beet_salad fried_calamari french_toast bread_pudding gyoza
beet_salad pad_thai peking_duck cheesecake beignets bibimbap poutine
beef_carpaccio eggs_benedict falafel french_onion_soup spaghetti_bolognese
poutine chicken_wings cheesecake cup_cakes prime_rib ceviche eggs_benedict
hot_dog beet_salad lobster_bisque frozen_yogurt macaroni_and_cheese pancakes
samosa lobster_roll_sandwich beef_tartare creme_brulee crab_cakes donuts
shrimp_and_grits tacos onion_rings grilled_salmon samosa hot_dog risotto
```

# 13 Conclusion

From the results, we can find that when we train our image set on the whole data, our validation accuracy increases to 70.09% while small subset realized about 65.31% validation accuracy. This means when we increase traning set with more decent class image, it will help a lot on training model. But there is trade off. While we have more data, the processing and training time is also increasing a lot. One of trick strategy to attack this is to use small representative subset to choose training architecture like image augumenation, final sequential layers, optimizer algorithms and their parameters and so on.

**Finall we ended up with 70.09% validation accuracy. This is big success, in fact, because we have 101 classes, thus, the random guess has less 1% accuracy, but after using pretrained models and adjusting some architecture, we can realize it to 70.09% from less 1%. This shows deep convolutional neural network is so powerful that classify 101 class food image in 101,000 huge dataset.**

During training, I found that if I added image augmentation, the validation accuracy acutally decreases this is becuase we have enough image data, so we don't need to rely on this technique to increase data size, and original image data is in good quality, better than augmented image data.

As for optimizer, I tried SGD and Adam, I found Adam is still performing better than SGD in image classification. Adam's recommended parameters, learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8, realized the best performance than ones that I manually changed.

As for final block of pretrained model, the interesting thing is that adding more dropout, linear layers will decrease the validaton accuracy. Only using one final linear layer performs better than adding one more layers. I think this is because pretrained models have sufficinet convolutional layers, dropout layers, and linear layers in the first half, so adding more these similar layers don't help at all.