Classification of Food Images Using Convolution Neural Networks

Prepared by:

Group 7

Joanne Chung

McLain Wilkinson

Ruonan Jia

Table of Contents

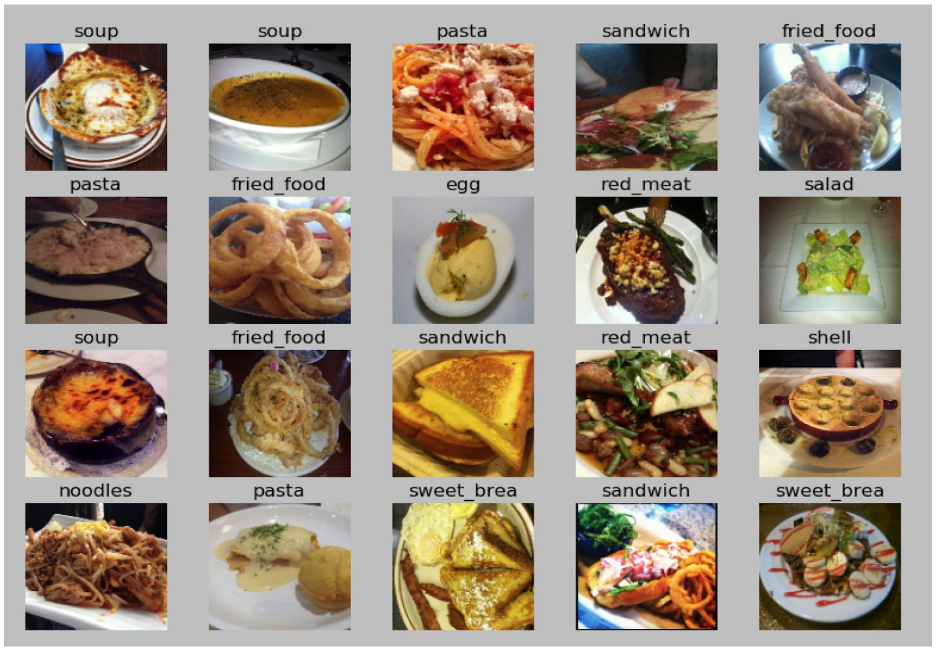
Introduction

Businesses such as Yelp utilize a system to display relevant pictures of menu items uploaded by users to accompany their text reviews of restaurants. The addition of these images to restaurant review pages adds value to the Yelp business model, giving other users visual depictions of menu items discussed in reviews. However, Yelp does not automatically label these images when they are uploaded, which can make it difficult for users to find pictures of specific food items. For instance, someone searching for pictures of a cheeseburger at a restaurant may have to scroll through many individual images until the desired picture shows up. This functionality can be inconvenient for a user. It would be desirable for Yelp to implement an image-labeling system so when a user searched for “cheeseburger” on a restaurant page, they would be able to see all pictures of cheeseburgers that other users had uploaded. For this project, we want to design and test a prototype of such a classification system. Using pictures of food items taken in a restaurant setting similar to images that a Yelp user may upload, we will attempt to train a convolution neural network (CNN) to classify them into twelve broad categories. The following sections of this report will provide an overview of each segment of the project process as well as our results and conclusions. We will begin by providing an overview of the data, followed by an outline of the experimental setup, explanation of the network and model design, finishing with a summarization of the results with conclusions from our findings.

Data Overview & Preprocessing

Data Overview

Our data is from Kaggle website and consists of 101,000 512x512 food images in 101 categories, with 1,000 images per category. The original data is contained in an “images” folder containing 101 subfolders titled by the food title. Each subfolder contains 1,000 .jpg images of the food described in the folder title. The below figure is shown twenty sample images. Comparing with the MNIST dataset, the food images are not clean and contain some amount of noise. For example, first two images are a Soup, contains a different color and background. Also, three images of the pasta category have totally different shapes, colors, textures and backgrounds. These discrepancies may have a huge impact on training a model and its ability to correctly predict image class.



Data Preprocessing

Before and during training a model, my team realized we need to pre-process data for finding a better CNN Model. My team uses five data preprocessing approaches.

1. HDF5 database
    a. The original data has 101,000 color jpeg images with ~512x512 pixels and also is reformatted as HDF5(Hierarchical Data Format) which is a database file and flat file. HDF5 is well-suited for huge amounts of matrix data and faster I/O than reading and converting individual jpeg images. However, the included HDF5 datasets include only 1000 total sample images so my team needed to create a new dataset from the included jpeg images.
2. Resize Image
    a. The original size is ~512x512 pixels. After creating images of several different sizes and comparing an accuracy while training a model, we found the best size to be 128x128 pixels, during training a model. This size is large enough for the network to discern important features from the images without taking up too much space and processing.
3. Combine classes
    a. We Combine classes to create 12 relatively broad classes comprised of visually similar food items. For instance, the “cake” category contains images from “carrot cake” and “red velvet cake”. Also, it adds to the total number of training images for each class, helping to give the network more examples to learn from.
4. New Dataset

a. The script create_dataset.py creates 2 HDF5 files entitled “food_train.h5” and “food_test.h5”, representing the training and testing image sets for the experiment. Combined, there are 48,000 total images representing 12 classes with each class having around 4000 examples. The two files are split using an 80/20 train/test split, stratified by class, and randomly ordered. The number of training images is 38,400 and number of testing images is 9,600.

Experimental Setup

With 48,000 total images spanning twelve categories, we divide the dataset into training and testing sets using an 80/20 training/testing split, stratified by target label and randomly ordered. The 38,400 training images are used to train a convolution neural network written in the PyTorch framework, the design of which is explained further in the following section. Using minibatches of size 40, the images are processed through the network to get output values. We calculate the performance of the network by comparing its output values to the corresponding target labels of the input images.

We have chosen to use Cross-Entropy loss to measure the performance of our classification model. For each image, the output of our network is a vector of length 12 containing values ranging from 0 to 1, the highest of which corresponds to the predicted class for the image. This performance function compares each value of the output vector to its corresponding value in the target vector, which is also a vector of length 12 containing 11 0’s and a single 1 at the location of the positive class label. The loss is calculated for each label by multiplying the log of each output probability by its target label (0 or 1) and summing the product to get the total loss for each image. This loss function penalizes errors proportionally to the confidence of the prediction. For instance, if the predicted probability for the class corresponding to the correct class was very small, the log loss will be very large. The loss value calculation for each image can be summarized by the following equation:

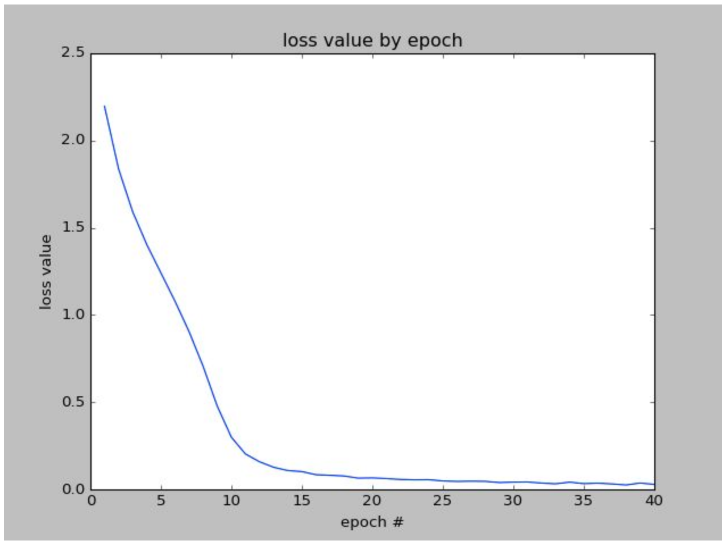
$$-\sum_{c=1}^{12} y_c \log(p_c)$$

where

y_c = binary indicator (0 or 1) for class c

p_c = predicted probability that observation is of class c

The calculated loss value is used to update the network parameters (weights of convolution kernels) by backpropagation. We use the Adam optimizer in PyTorch with an initial learning rate of 0.001 to update the parameters after the processing of each minibatch. The previous steps are completed for each minibatch until every image in the database is processed by the network. We complete 30 passes of the training set. By plotting the average loss value for each epoch as seen below, we see that 30 epochs is a sufficient number of passes of the training set to achieve a stabilized loss value as well as a reasonable resulting network accuracy.



The loss begins to level off near epoch 25, and continuing to make passes over the training images after this epoch does not produce a significantly lower average loss value, so 30 epochs is a reasonable number of training epochs to perform. After 30 epochs, the network will have been sufficiently trained on the training images.

Finally we test the performance of the network using test images, which the network did not encounter during training. Again using minibatches of size 40, the test images are processed by the network to get an output. We compare the outputs to the target labels, tracking the number of "correct" predictions. By dividing this number by the total number of test images, we will have the overall classification accuracy of the network on the test images. In addition, we sum the predicted output classes for each individual class to create a confusion matrix and calculate the network accuracy for individual classes, allowing us to better understand the performance of the network, highlighting its strengths and shortcomings.

Due to the amount of time it takes to train the network (~2 hours, 40 minutes for 30 training epochs) we are unable to conduct many trials to see if the output of the network is consistent or changes significantly for different trials. Ideally we would be able to compare the results of multiple trials to understand if the network strengths and shortcomings are consistent.

Network / Model Design

There is no generalized solution for designing a CNN. The design depends on the purpose of our project, like input characteristics, accuracy, training time, adaptation, computing resources ... We are focused on getting the highest accuracy. To begin, we approach the problem in an empirical way: initialize the network parameters with a lot of different numbers and adjust them to see what improves accuracy.

Things in model that affect our accuracy:

Number of Layers:

Number of layers	2 layers, kernel size 3	4 layers, kernel size 3	6 layers, kernel size 3
Accuracy	6%	~40%	60%

With a data size of [48000, 3, 128, 128] (48,000 images of 3 color channels of 128x128 pixels), more than 6 layers will give us the best accuracy. After the 6th Convolution/Max Pooling layer, we reach the final output size of 2x2, a significant reduction in size from the 128x128 input images. Achieving 60% accuracy on the test set is a big improvement, so we decided that 6 layers are necessary in our model.

Kernel Size: With a small kernel size, the kernel can capture the outline of the images better. Imagining we have a very big kernel size, when the kernel slide on the picture, we may miss some features. With an 128x128 input size, we decided to try kernel size 5x5 and 10x10 at the first four layers, 4x4 at 5th layer and 2x2 at 6th layer. The reason we use a much smaller kernel size at the last two layers is because size of the feature map is getting smaller by going through the MaxPooling layer (with a kernel size 2x2, stride=2, padding=1, the size of feature map after pooling is half of the input size from convolution layer). Kernel size 5 gets slightly higher accuracy than kernel size 10. We decided to use kernel size 10, which is able to see the pattern of feature map better than smaller kernel size.

The calculation of output size is a big part for designing a model by Pytorch.
We are using MaxPooling layer with a kernel size 2x2, stride=2, padding=1, which can give us the half of the input size from convolution layer. We want our output size from each layer is 64x64, 32x32…. To make this happen, we need to make the output size from convolution layer close to the input size from each layer. Then we decided to use padding=4 and stride =1, so we keep almost the same size as the input size.

See the example calculations as below:
Equations to calculate output size after convolution:

$w = (w + 2*PAD - \text{kernel size}) / STRIDE + 1$

h = (h + 2*PAD−kernel size) / STRIDE +1

1st Convolution layer(kernel size=10, padding=4, stride=1)
1st MaxPolling layer(kernel size=2, padding=1, stride=2)

Conv1: w=(128+2*4-10)/1+1=127 h=(128+2*4-10)/1+1=127
Pool1: w=(127+1)/2=64 h=(127+1)/2=64

We use the same kernel size, padding size and stride size at the first 4 layer.
So the output size for each layer:
128 → 64 → 32 → 16 → 8
From the 5th layer, with an input size 8x8, we cannot use kernel size larger than 8. We picked kernel size as 4, at the same time, padding should not be bigger than kernel size.

5th Convolution layer (kernel size=4, padding=2, stride=1)
5th MaxPooling layer (kernel size=2, padding=1, stride=2)

Conv5: w=(8+2*2-4)/1+1=9 h=(8+2*2-4)/1+1=9
Pool5: w=(9+1)/2=5 h=(9+1)/2=5

6th Convolution layer(kernel size=2, padding=0, stride=1)
6th MaxPooling layer(kernel size=2, padding=0, stride=2)

Conv6: w=(5+2*0-2)/1+1=4 h=(5+2*0-2)/1+1=4
Pool6: w=(4+0)/2=2 h=(4+0)/2=2

Number of Kernels:
For a better result, we are more willing to capture enough main features from the original input. Theoretically, we need a big number (like 256) as our first layer number of kernels. However, in reality if we put too many inputs at the first layer, when we use the cloud, we get an error that the gpu will be out of memory; even though we tried to put the batch size smaller it will be worse and time cost is too high. The highest number we can put at first layer is 128. To solve this problem, we decided to start with a reasonable number of kernels at the first layer (like 64), then increase the number of kernels in the following 3 layers and start decreasing the number of kernels at the last 2 layers (so we won't have too many neurons, also during that time for each feature map a smaller number of kernels will be enough to capture the main feature). The number of kernels used for each layer is 64, 128, 256, 394, 256, 128. While the total number of feature maps created will be increasing, we also tried to have a decreasing list of number of

kernel for each layer. The highest number we can start with is 128, so we tried 128, 128, 128, 128, 128, 64. The accuracy is 4% less than our model.

Dropout layer:
With the new dataset having around 4,000 images per class and 48,000 total images, it is difficult to have underfitting in neural network model. Overfitting happens when the model begins to track variance in the data, so we added dropout on the fully connected layer with 0.5 dropout ratio (drop half of the neurons) to prevent overfitting in our CNN model.

SoftMax function at last layer:
Our goal is to do images classification for 12 classes. For more than 2 classes classification, SoftMax function suits our model. The output of the model is a vector of length 12 producing values ranging from 0 to 1, representing the probability that the observation is of a certain class. The class with the highest such probability is the predicted output of the network.

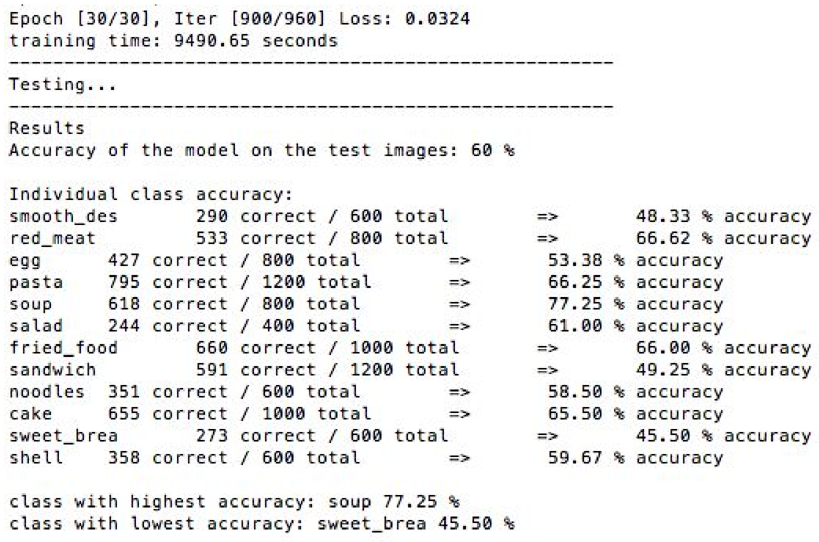
Adam optimizer:
We used this training function because it showed better performance than SGD. SGD uses a static learning rate, while Adam is using dynamic learning rate. When the optimizer reaches the local minimum, the momentum will make it keeping going and find the global minimum. Adam will work better for a complicated neural network like CNN.

Epochs:
For the first trial, we pick an epoch number as 50, which cost us 5 hours to train the model. The training loss is very low from 20 epochs.

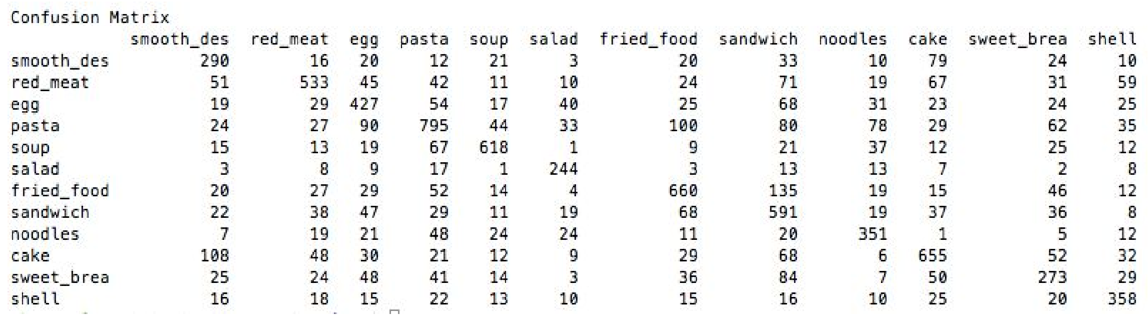
Model Summary
Pytorch Framework
6 Convolution/MaxPooling layers & 1 Fully Connected Layer with dropout layer
Number of layers & size of kernels limited by input image size
Kernel size = 10 for first 4 layers, 4 for 5th layer, 2 for last convolution layer
Numbers of kernels: ranging from 64 to 384 for each layer
Batch size: 40
Epochs: 25
Cross Entropy Loss
Adam optimizer

Results

The usefulness of our CNN can be determined by its ability to accurately label previously unseen test images. Upon training the network, we obtained the results of the CNN's predicted outputs for the 9,600 images in our testing dataset. The model will print out overall results containing the following metrics: Overall accuracy (%) on the testing set, individual class accuracy, the two categories producing the maximum and minimum accuracy values, and a confusion matrix. The model output and confusion matrix are shown below.

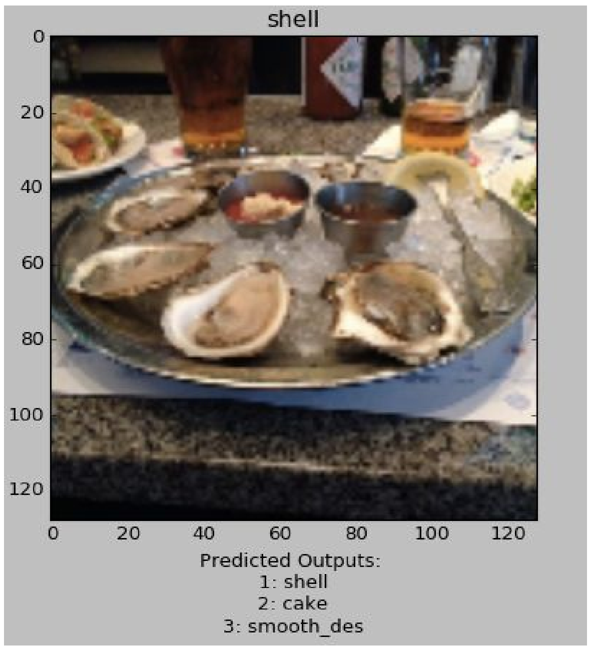


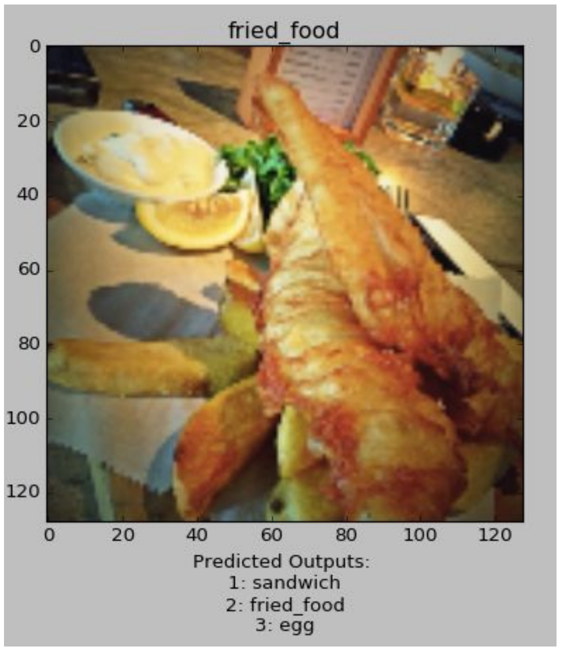
```
Epoch [30/30], Iter [900/960] Loss: 0.0324
training time: 9490.65 seconds
-------------------------------------------------------
Testing...
-------------------------------------------------------
Results
Accuracy of the model on the test images: 60 %

Individual class accuracy:
smooth_des       290 correct / 600 total        =>       48.33 % accuracy
red_meat         533 correct / 800 total        =>       66.62 % accuracy
egg      427 correct / 800 total        =>       53.38 % accuracy
pasta    795 correct / 1200 total       =>       66.25 % accuracy
soup     618 correct / 800 total        =>       77.25 % accuracy
salad    244 correct / 400 total        =>       61.00 % accuracy
fried_food       660 correct / 1000 total       =>       66.00 % accuracy
sandwich         591 correct / 1200 total       =>       49.25 % accuracy
noodles  351 correct / 600 total        =>       58.50 % accuracy
cake     655 correct / 1000 total       =>       65.50 % accuracy
sweet_brea       273 correct / 600 total        =>       45.50 % accuracy
shell    358 correct / 600 total        =>       59.67 % accuracy

class with highest accuracy: soup 77.25 %
class with lowest accuracy: sweet_brea 45.50 %
```



```
Confusion Matrix
             smooth_des  red_meat  egg  pasta  soup  salad  fried_food  sandwich  noodles  cake  sweet_brea  shell
smooth_des          290        16   20     12    21      3          20        33       10    79          24     10
red_meat             51       533   45     42    11     10          24        71       19    67          31     59
egg                  19        29  427     54    17     40          25        68       31    23          24     25
pasta                24        27   90    795    44     33         100        80       78    29          62     35
soup                 15        13   19     67   618      1           9        21       37    12          25     12
salad                 3         8    9     17     1    244           3        13       13     7           2      8
fried_food           20        27   29     52    14      4         660       135       19    15          46     12
sandwich             22        38   47     29    11     19          68       591       19    37          36      8
noodles               7        19   21     48    24     24          11        20      351     1           5     12
cake                108        48   30     21    12      9          29        68        6   655          52     32
sweet_brea           25        24   48     41    14      3          36        84        7    50         273     29
shell                16        18   15     22    13     10          15        16       10    25          20    358
```

The above outputs show a 60% overall accuracy value on the testing set, suggesting the model predicted the correct label for 60% of images contained in the test set. While this accuracy is not stellar, it is a significant improvement over the accuracy one would expect to achieve from a random guess, a 1/12 chance (~8-9%). The following breakdown of the CNN's predictions for images of individual classes provides more insight on the behavior of the network and helps to explain what could have lowered the overall accuracy. The network achieved the best accuracy labeling images belonging to the "soup" category, correctly identifying 77.25% of soup images, while only correctly labeling 45.5% of "sweet breakfast" items. This difference in accuracy percentage (31.75%) for these two categories of items is significant, and potentially suggests issues concerning the uniformity and similarity of images in the "sweet breakfast" category. The confusion matrix offers an even deeper breakdown of the CNN's test set predictions, showing the specific counts of predicted labels for each image category with columns representing the actual label and rows representing the predicted label. The diagonal values show correctly classified observations, and all other cells represent incorrectly classified images. Consequently, if non-diagonal count is particularly large, it may suggest that the network struggled to distinguish the two classes. For instance, one of the highest counts not occurring on the diagonal represents "smooth dessert" images misclassified as "cake". Interestingly, the most frequently mislabeled "cake" image was predicted to be in the "smooth dessert" class. So the CNN seems to have difficulty distinguishing images of these two classes. Additionally, the highest number of misclassifications occurs for "sandwich" images misclassified as "fried food". Inspecting some of the images belonging to these few categories does provide intuitive explanations for the errors, however. The final results the model produces involve plotting 10 labeled images along with the top three most likely labels predicted by the CNN. Two sample images with predicted label plots are shown below.





The CNN correctly predicted the "shell" image on the left to be in the "shell" category. Unfortunately, the "fried food" image on the right was incorrectly predicted to be a "sandwich", although "fried food" was the second most likely prediction. The food item in this image does appear to be sandwich-like upon first glance, so the network didn't make a terrible prediction. Other images prove to be more problematic. For example, some "sandwich" images in the dataset contain a picture of a sandwich and french fries on a plate. What should the true class for this picture be, "sandwich" or "fried food"? Here is where the network as it currently exists may fall short at classifying images correctly. For future consideration, we may want to allow images to carry multiple labels, so the sandwich and fries image could be correctly identified as both "sandwich" and "fried food".

Summary & Conclusions (McLain)

Given the structure and accompanying difficulty of our problem, we successfully built a network that could achieve a reasonable accuracy classifying images into 12 broad categories of food. The biggest problems of our experiment are more associated with the nature of the images in our dataset and our decisions concerning their labels. While combining classes together may have provided the network with more training images, it introduced much more variance into the nature of the categories. For instance, our "pasta" category contains images consisting if 6 very diverse dishes in shape and color: gnocchi, lasagna, macaroni and cheese, ravioli, spaghetti bolognese, and spaghetti carbonara. Even though we had more training images, the network was not given very uniform examples of what a pasta dish entails. Maybe this combining did more harm than good.

To prevent this problem, we could introduce using multiple labels for each image. For instance, a picture of a slice of cake topped with ice cream could be labeled by a tuple (0, 9) representing both "smooth dessert" and "cake". This way, the network would not be penalized for classifying the network as only one or the other. Additionally, we could allow the network to classify an image with 2 classes, by selecting both classes if the predicted probability for each is greater than a certain threshold. This could allow the network to predict "sandwich" AND "fried food" if the image contains pictures of both.

Either way, the problem could be worked through with the presence of cleaner and more uniform data. Maybe our network would have been more successful had we not chosen to combine classes of images.

References

Cross-entropy loss:
https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html#cross-entropy

Data:
https://www.kaggle.com/kmader/food41

Python Script for illustrating CNN
https://github.com/yu4u/convnet-drawer/blob/master/convnet_drawer.py

Appendix: Code for Model (food_CNN.py)

```
import torch
import torch.nn as nn
from torch.autograd import Variable
import torchvision.datasets as dsets
import matplotlib.pyplot as plt
import h5py
import numpy as np
import time
import pandas as pd
pd.set_option('display.width', 320)
pd.set_option("display.max_columns", 15)
# ---------------------------------------------------------------------------------
# training parameters
batch_size = 40
num_epochs = 30

# train and test set files
h5train_file = "food_train.h5"
h5test_file = "food_test.h5"

# class for creating torch dataset from HDF5 database
class DatasetFromHdf5(torch.utils.data.Dataset):
   def __init__(self, file_path):
      super(DatasetFromHdf5, self).__init__()
      hf = h5py.File(file_path)
      self.data = hf.get('images')
      self.target = hf.get('labels')
      self.classes = hf.get('categories')

   def __getitem__(self, index):
      return torch.from_numpy(self.data[index, :, :, :].T).float(), self.target[index]

   def __len__(self):
      return self.data.shape[0]


# create datasets and data loaders
train_dataset = DatasetFromHdf5(h5train_file)

test_dataset = DatasetFromHdf5(h5test_file)

train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# print training/testing dataset sizes and target classes
target_classes = tuple(target_class.decode() for target_class in train_dataset.classes)
```

```
print('Number of training images:', train_dataset.__len__())
print('Number of testing images:', test_dataset.__len__())
print('target classes:', target_classes)
print('---------------------------------------------------')
# ------------------------------------------------------------------------------
# define the network model
class CNN(nn.Module):
  def __init__(self):
    super(CNN, self).__init__()
    self.layer1 = nn.Sequential(            # input size = 128 x 128
      nn.Conv2d(3, 64, kernel_size=10, padding=4),
      nn.BatchNorm2d(64),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2, padding=1))  # output size = 64 x 64
    self.layer2 = nn.Sequential(
      nn.Conv2d(64, 128, kernel_size=10, padding=4),
      nn.BatchNorm2d(128),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2, padding=1))  # output size = 32 x 32
    self.layer3 = nn.Sequential(
      nn.Conv2d(128, 256, kernel_size=10, padding=4),
      nn.BatchNorm2d(256),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2, padding=1))  # output size = 16 x 16
    self.layer4 = nn.Sequential(
      nn.Conv2d(256, 384, kernel_size=10, padding=4),
      nn.BatchNorm2d(384),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2, padding=1))  # output size = 8 x 8
    self.layer5 = nn.Sequential(
      nn.Conv2d(384, 256, kernel_size=4, padding=2),
      nn.BatchNorm2d(256),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2, padding=1))  # output size = 5 x 5
    self.layer6 = nn.Sequential(
      nn.Conv2d(256, 128, kernel_size=2),
      nn.BatchNorm2d(128),
      nn.ReLU(),
      nn.MaxPool2d(2, stride=2))           # output size = 2 x 2
    self.dropout = nn.Dropout(0.5)
    self.fc = nn.Linear(2 * 2 * 128, 12)

  def forward(self, x):
    out = self.layer1(x)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)
    out = self.layer5(out)
    out = self.layer6(out)
    out = self.dropout(out)
```

```
        out = out.view(out.size(0), -1)
        out = self.fc(out)
        return out

# ---------------------------------------------------------------------------------
# initialize the model
cnn = CNN()
cnn.cuda()
# ---------------------------------------------------------------------------------
# Loss and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(cnn.parameters(), lr=0.001)
# -----------------------------------------------------------------
# Train the Model
num_images = train_dataset.__len__()
start = time.time()
epoch_list = [e + 1 for e in range(num_epochs)]
loss_list = []
print('Training...')

for epoch in range(num_epochs):
    epoch_loss = []
    for i, (data, target) in enumerate(train_loader):
        images = Variable(data).cuda()
        labels = Variable(target).cuda()

        # Forward + Backward + Optimize
        optimizer.zero_grad()
        outputs = cnn(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        if (i + 1) % 100 == 0:
            print('Epoch [%d/%d], Iter [%d/%d] Loss: %.4f'
                  % (epoch + 1, num_epochs, i + 1, num_images // batch_size, loss.item()))
        epoch_loss.append(loss.item())

    # add average loss for the epoch to list of average loss values
    loss_list.append(np.mean(epoch_loss))

end = time.time()

print("training time:", '%.2f' % (end - start), 'seconds')
print('--------------------------------------------------------')
# ---------------------------------------------------------------------------------
# Test the Model
cnn.eval()  # Change model to 'eval' mode (BN uses moving mean/var).

print('Testing...')
```

```
correct = 0
total = 0
correct_array = np.zeros(len(target_classes))
total_array = np.zeros(len(target_classes))
confusion = {x: [0 for i in range(len(target_classes))] for x in range(len(target_classes))}

for images, labels in test_loader:
   input_images = Variable(images).cuda()
   outputs = cnn(input_images)
   _, predicted = torch.max(outputs.data, 1)
   total += labels.size(0)
   correct += (predicted.cpu() == labels).sum()
   for pred, actual in zip(predicted.cpu(), labels):
       confusion[int(actual)][int(pred)] += 1
       total_array[actual] += 1
       if pred == actual:
           correct_array[actual] += 1
   images = images
# -------------------------------------------------------------------------------------
# display overall results
print('------------------------------------------------------')
print('Results')
print('Accuracy of the model on the test images: %d %%' % (100 * correct / total))
print('')
# display individual category results
print('Individual class accuracy:')
target_classes = tuple(target_class.decode() for target_class in train_dataset.classes)
for i, t in enumerate(target_classes):
   print(t + '\t', int(correct_array[i]), 'correct', '/', int(total_array[i]), 'total',
         '\t=>\t', '%.2f' % (100 * correct_array[i] / total_array[i]), '%', 'accuracy')

# show and print out class/accuracy for highest and lowest accuracy values
pct_array = correct_array / total_array
best = int(np.argmax(pct_array))
worst = int(np.argmin(pct_array))
print('')
print('class with highest accuracy:', target_classes[best],
    '%.2f' % (100 * correct_array[best] / total_array[best]), '%')
print('class with lowest accuracy:', target_classes[worst],
    '%.2f' % (100 * correct_array[worst] / total_array[worst]), '%')
print('')

# create and show confusion matrix
confusion = pd.DataFrame.from_dict(confusion)
confusion.columns = target_classes
confusion.index = target_classes
print('Confusion Matrix')
print(confusion)
```

```
# plot average loss for each epoch
plt.figure(0)
plt.plot(epoch_list, loss_list)
plt.title('average loss by epoch')
plt.xlabel('epoch #')
plt.ylabel('average loss value')

# show 10 test images with target class & top 3 predicted outputs
for i, (output, label) in enumerate(zip(outputs.data.cpu(), labels)):
    if i < 10:
        output = np.array(output)
        ndx = output.argsort()[-3:][::-1]
        caption = 'Predicted Outputs: \n'
        for j, n in enumerate(ndx):
            caption = caption + (str(j + 1) + ': ' + target_classes[n] + '\n')
        image = np.array(images[i]).T
        plt.figure(i + 1)
        plt.imshow(image)
        plt.title(target_classes[label])
        plt.xlabel(caption)
        plt.tight_layout()

# show loss plot & 10 images w/ predicted classes
plt.show()
# ------------------------------------------------------------------------------------
# Save the Trained Model
torch.save(cnn.state_dict(), 'cnn.pkl')
# ------------------------------------------------------------------------------------
```