D

Design Documentation

Data Science 440
Design Document

Team Six

Juliana(Jiayin) Hu, Sydney Wehn, Kangdong Yuan, Xueqing Zhang, Yuqi Gao

Kmean and visualization (contributed by Kangdong Yuan)

You can access the code in following link: https://github.com/yedkk/kmean-model

The program I made is to input the pictures of each ancient Egyptian alphabet into the program, and the final output is the label of each picture.

In this plan, I think jupyter notebook using python language is a better choice. Because jupyter notebook can easily add comments and instructions, and jupyter notebook running code is divided into different code blocks. So users can execute code block of different functions separately without wasting time to execute all the codes.

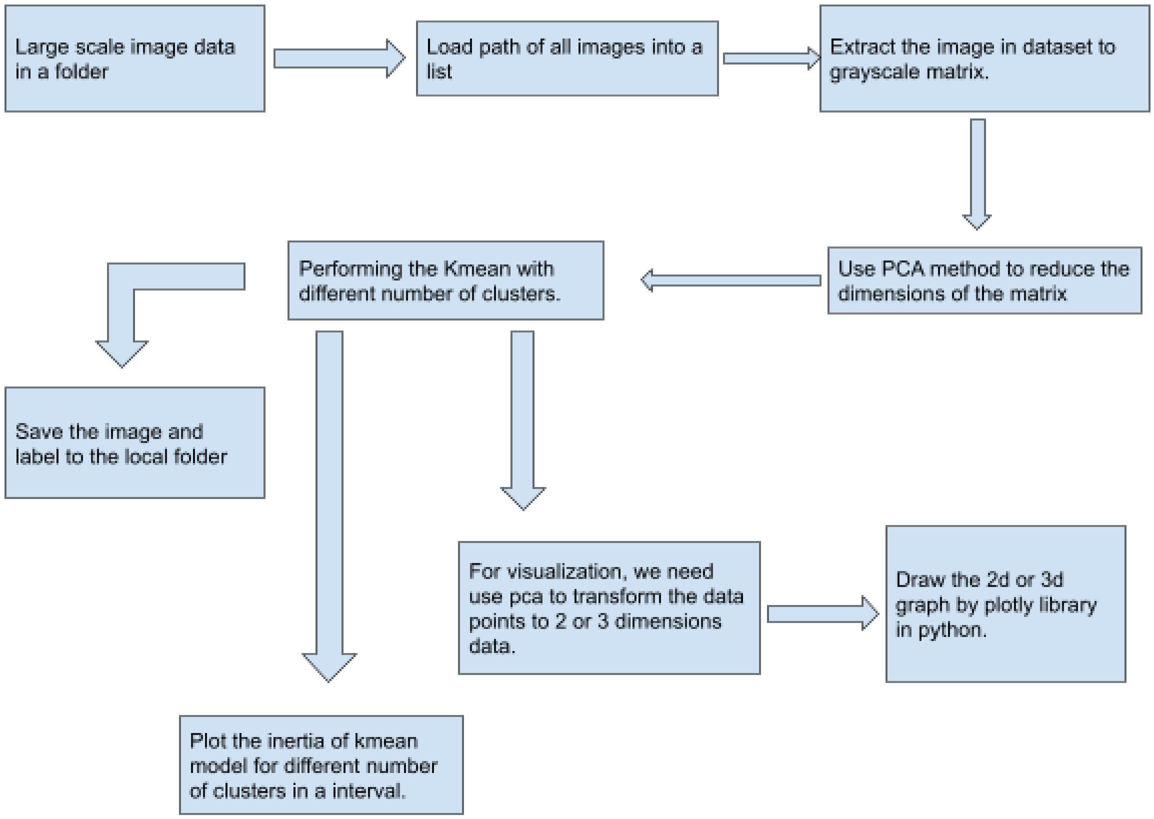
There are 6 steps in clustering images:
First, if we want to classify the letters of the object pictographs for unsupervised learning, we need a certain amount of data to ensure the stability of the k-means model. In this database, we have more than four thousand different pictures to train the model. Second, we need to enter the location of the folder where the pictures are placed, and use a for loop to add the paths of all pictures to a list. Third, we need to use a for loop to extract the feature values of all pictures into a list, and then convert this list into a numpy matrix. Fourth, we need to use the PCA method to reduce the dimensionality of the matrix to speed up our training model. Fifth, using the k-means method to train the model, users can customize the number of clusters they want. Sixth, the labels are output in the form of file names, and the clustered pictures are saved locally.

Finally, I made detailed settings for these steps and added notes at each step. I set each step as a code block, and each parameter is saved as a variable. The user can easily change the variable value to adjust the k-means model and PCA function.

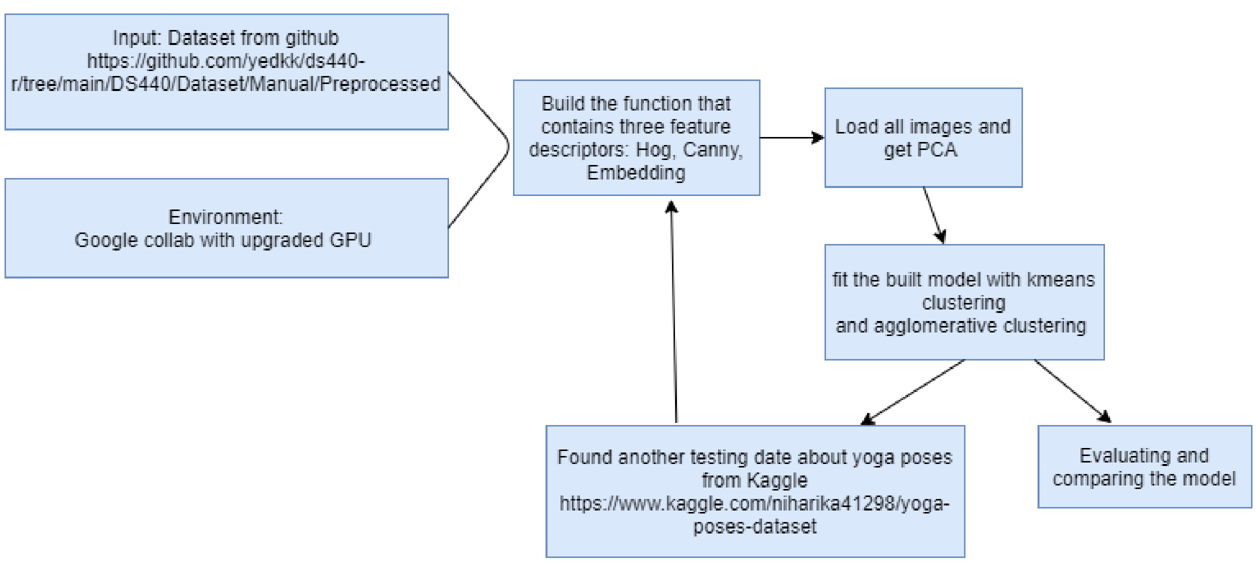
If the user wants to see the error value of the k-means model, when the number of clusters is within an interval. I set up a code block to automatically execute this process. The user only needs to specify a range of the number of clusters, and the code can automatically generate a line graph.

For the visualization of the k-means model, I defined two code blocks to generate 2D and 3D images. The PCA transformation of the data is automatically executed in these code blocks so that each data point can be accurately projected in two-dimensional and three-dimensional space.

Users only need to set the number of clusters to automatically generate two-dimensional and three-dimensional images of the k-means model.



Clustering with Feature Descriptor



ResNet34 and AlexNet Classification (contributed by Sydney)

You can access the code in following link:
https://colab.research.google.com/drive/1jqP8RmVEeXl0ATujQFmOWH8JQOImE5w7?usp=sharing

As an overview, the program inputs images associated with the ancient Egyptian alphabet, and the final output is the label of each picture classified by the ResNet34 and Alexnet. The 27 classes all represent letters in the Gardiner Sign list which incorporate different types of animal, object, and human categories as seen here https://en.wikipedia.org/wiki/Gardiner%27s_sign_list.

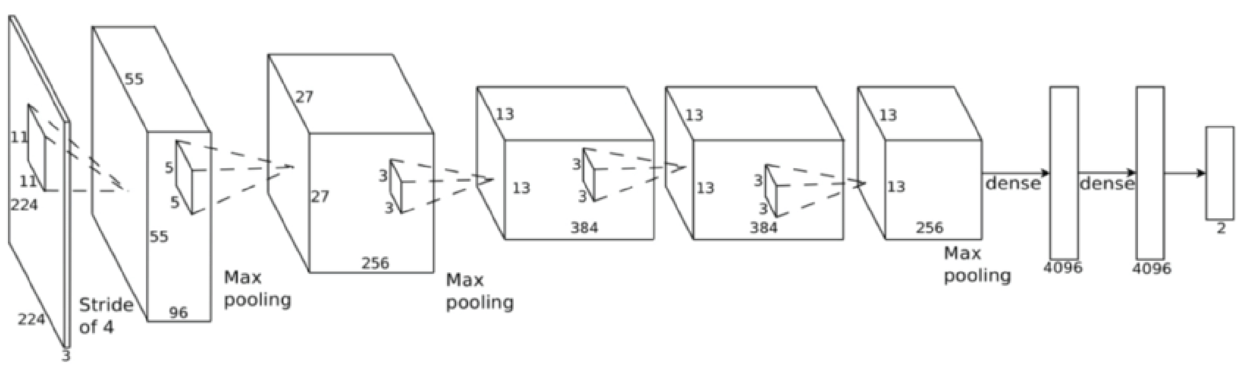
In more detail, I first downloaded the image data from the GitHub repository by copying the zip file into my Google Drive. I then manually created folders representing the Gardiner Signs and dragged the images into their respective folders. I then used Google Colab Pro to pull in the data from the train, validation, and test sets.

I then went ahead to transform the image data using standard deviations and mean values with RandomResizedCrop, RandomRotation, RandomHorizontalFlip, ToTensor, and Normalize on both the training and validation sets. Then I created data loaders using torch tools and loaded in the pretrained AlexNet and ResNet34 models. From there, I ensured that I was classifying to 27 classes in both models and used loss and optimizer functions to train the models.

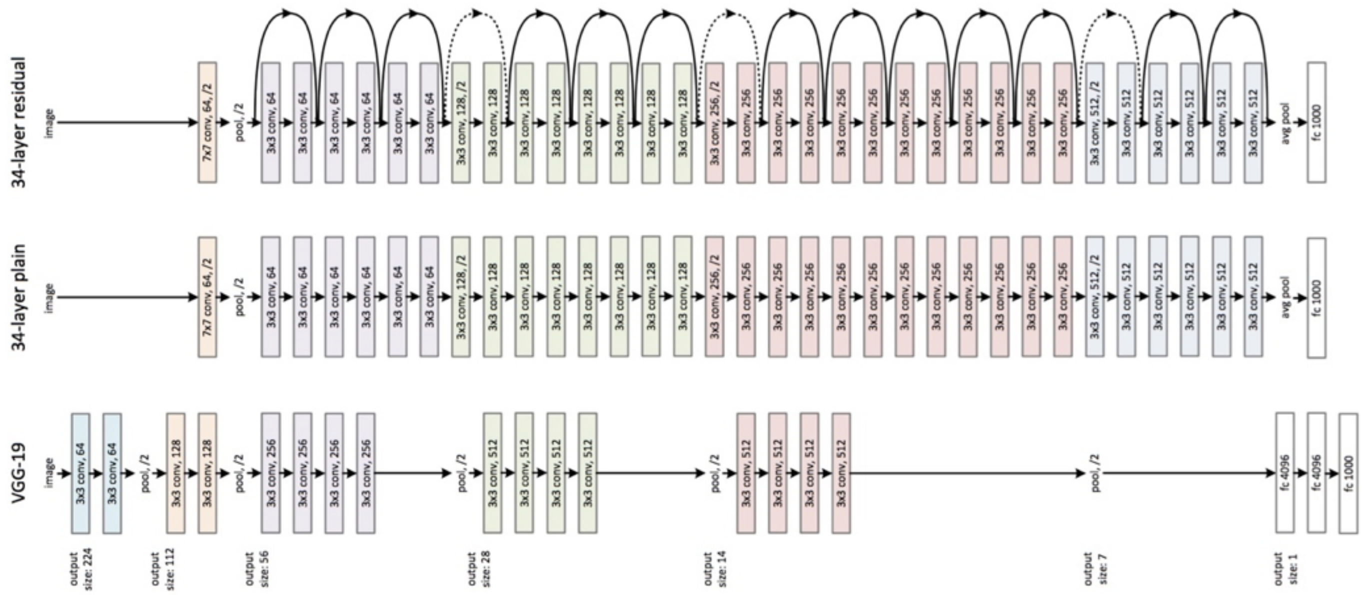
To evaluate the performance of the models, I used train and validation losses as well as confusion matrices to give insights into precision, recall, and accuracy.

To give a better idea as to how ResNet34 and AlexNet layers work, the two architectures of layers are seen below.

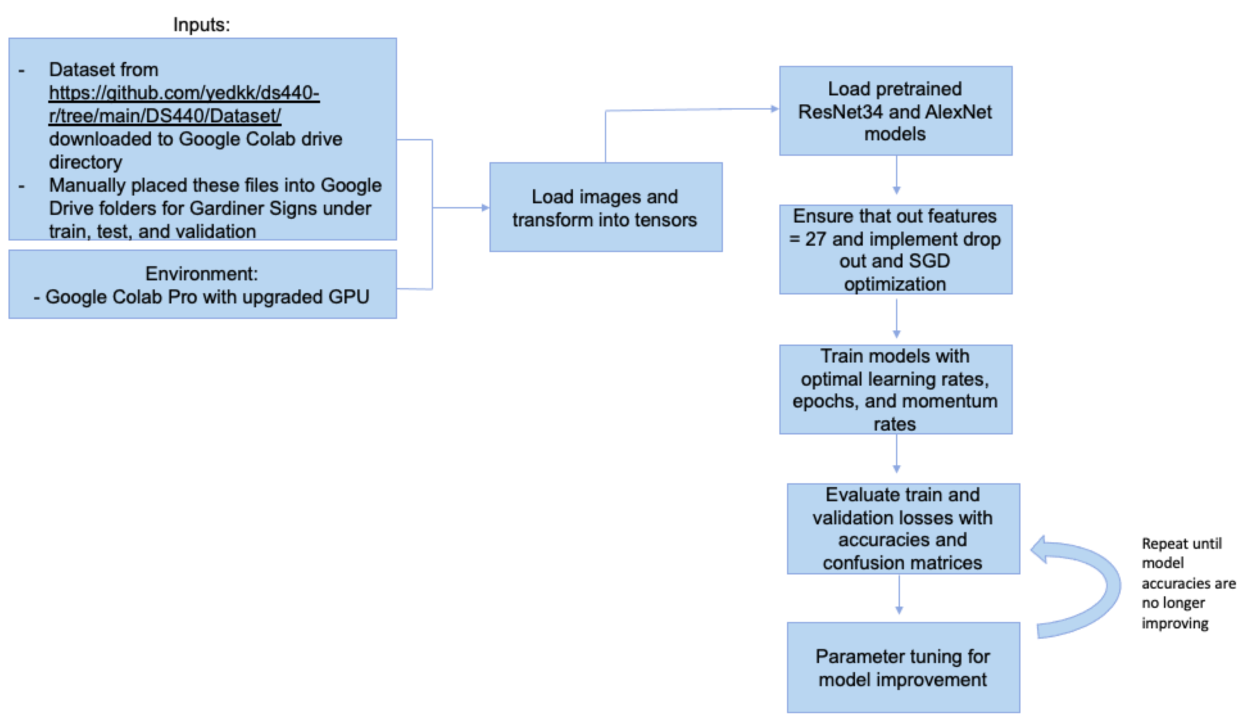
AlexNet Architecture:



ResNet34 Architecture:



The steps used in this design are as follows.



Appendix:

https://github.com/yedkk/kmean-model
https://github.com/yedkk/ds440-r/blob/main/DS440/capclass.py

Clustering with Feature Descriptor :

```
#encoding=utf-8
import numpy as np
import cv2
from skimage import feature as ft
import matplotlib.pyplot as plt
import torchvision.models as models
import pretrainedmodels
import pretrainedmodels.utils as utils
import torch
import os, sys
import json
from sklearn.decomposition import PCA

load_img = utils.LoadImage()

def getHOG(imfn):
    img = cv2.imread(imfn)
    img = cv2.resize(img, (64,64))
    features = ft.hog(img, orientations=6, pixels_per_cell=[8, 8], cells_per_block=[2, 2], visualize=True)
    return features[1]
    # plt.imshow(features[1], cmap=plt.cm.gray)
    # plt.show()


def getCanny(imfn):
    img = cv2.imread(imfn)
    im = cv2.resize(img, (64, 64))
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    features = ft.canny(im)
    return features
    # plt.imshow(edges1, cmap=plt.cm.gray)
    # plt.show()

def getImageEmbedding(imfn):
    import os
    os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

    model_name = 'inceptionv3'
    model = pretrainedmodels.__dict__[model_name](num_classes=1000, pretrained='imagenet')

    tf_img = utils.TransformImage(model)
    input_img = load_img(imfn)
    input_tensor = tf_img(input_img)  # 3x400x225 -> 3x299x299 size may differ
    input_tensor = input_tensor.unsqueeze(0)  # 3x299x299 -> 1x3x299x299
    input = torch.autograd.Variable(input_tensor,
                                    requires_grad=False)
```

```
    output_features = model.features(input)
    emb = output_features[0].detach().numpy()
    return emb

imgfn = "3/030000_S29.png"

def getAllImage():
    sons = os.listdir()
    image_fn_list = []
    for son in sons:
        name = os.path.join("./", son)
        if os.path.isdir(name):
            for fn in os.listdir(name):
                tmp_fn = os.path.join(name, fn)
                if 'png' in tmp_fn:
                    image_fn_list.append(tmp_fn)
    return image_fn_list


def getEmb():
    image_fns = getAllImage()
    fw = open('emb.txt', 'w')
    for fn in image_fns:
        hog_features = getHOG(fn).flatten()
        canny = getCanny(fn).flatten()
        inception_emb = getImageEmbedding(fn).flatten()
        data = {
            'hog': hog_features.tolist(),
            'canny': canny.tolist(),
            'inception:': inception_emb.tolist()
        }
        fw.write(fn + "_" + json.dumps(data) + '\n')
        # res_dict[fn] = data
    fw.close()


def getPCA():
    lines = open('emb.txt').readlines()
    fn_list = []
    hog_list = []
    canny_list = []
    inception_list = []
    for line in lines:
        res = line.strip().split('_')
        fn = "_".join(res[:-1])
        # print(res[-1])
        data = json.loads(res[-1])
        fn_list.append(fn)
        hog_list.append(data['hog'])
        canny_list.append(data['canny'])
        inception_list.append(data['inception:'])

    hogs = np.array(hog_list, dtype=np.float32)
    pca_1 = PCA(n_components=64)
```

```
    hogs_pca = pca_1.fit_transform(hogs)
    print('===hog done.')

    canny = np.array(canny_list, dtype=np.float32)
    pca_2 = PCA(n_components=64)
    canny_pca = pca_2.fit_transform(canny)
    print('===canny done.')

    inception = np.array(inception_list, dtype=np.float32)
    pca_3 = PCA(n_components=1024)
    inception_pca = pca_3.fit_transform(inception)
    print('===inception done.')

    fw = open('pca.txt', 'w')
    for i in range(len(fn_list)):
        fn = fn_list[i]
        data = {
            'hog': hogs_pca[i].tolist(),
            'canny':canny_pca[i].tolist(),
            'inception': inception_pca[i].tolist()
        }
        fw.write(fn + '_' + json.dumps(data) + '\n')
    fw.close()

def cluster():
    f = open('pca.txt')
    fn_list = []
    emb_list = []
    for line in f:
        res = line.strip().split('_')
        fn = "_".join(res[:-1])
        data = json.loads(res[-1])
        emb_list.append(data['hog'] + data['canny'] + data['inception'])
        fn_list.append(fn)

    emb = np.array(emb_list, dtype=np.float)
    from sklearn.cluster import KMeans
    clf = KMeans(n_clusters=100)
    y = clf.fit_predict(emb)
    fw = open('kmeans.txt', 'w')
    for i in range(len(fn_list)):
        fw.write(fn_list[i] + ':' + str(y[i]) + '\n')
        # print(y[i], fn_list[i])
    fw.close()

    from sklearn.cluster import AgglomerativeClustering
    y = AgglomerativeClustering(linkage='ward', n_clusters=100).fit_predict(emb)
    fw = open('agg_cluster.txt', 'w')
    for i in range(len(fn_list)):
        fw.write(fn_list[i] + ':' + str(y[i]) + '\n')
        # print(y[i], fn_list[i])
    fw.close()
    # for i in range(len(fn_list)):
    #     print(y[i], fn_list[i])
```

```
def run():
    cluster()
    # getPCA()
run()
# getImageEmbedding(imgfn)
# getCanny(imgfn)
# getHOG(imgfn)
```